Effective Software Development Series

Scott Meyers, Consulting Editor



Effective C#

50 Specific Ways to Improve Your C#

Second Edition

Bill Wagner

目錄

Effective C# 改善C#程序的50种方法

作者：Bill Wagner

译者：DSQiu

来源：Effective C# 中文版改善C#程序的50种方法 第二版

第一章 C# 语言习惯

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

为什么程序已经能正常工作了，你还要改变它呢？答案就是你相信可以让它变得更好。你改变工具或语言，因为你可以变得更富有成效。如果你不改变你的习惯，你将不会受到预期的效果。对于C#这种和我们已经熟悉的语言，如 C++ 或 Java 有诸多共通之处的新语言，情况更是如此。 C# 是另一个强大的语言，很容易陷入其他语言的语言习惯。这将阻止你掌握最有效的 C# 。 C# 语言从2001年发布的第一个商业化版本以来一直在演变。相比早期的版本，它变得更加远离 C++ 或 Java 。如果你是从另一种语言转到 C# 语言的，你需要学习 C# 的习惯用法以至于语言为你所有，而不是阻碍你。本章讨论你应该改变的习惯并且你需要做什么。

小结：

附上第一章的目录：

从字面上看，感觉第一章主要介绍的是 C# 的语法，但是D.S.Qiu 重点应该是本书或者是 C# 强调的规范或编程模式。

D.S.Qiu

Effective C# 中文版改善C#程序的50种方法 第二版

Effective C# : 50 Specific Ways to Improve Your C# 2nd Edition

原则1：使用 属性（Poperty）代替可直接访问的数据成员 （Data Member）

By D.S.Qiu

尊重他人的劳动， 支持原创，转载请注明出处：**http://dsqiu.iteye.com**

在 C# 语言里，属性已经是“第一类公民（first-class citizens）”。自从C#1.0的发布以来，多个强化，使得属性有了更多的表现力。你可以使用 getter 和 setter 函数指定不同的访问限制。使用隐式（implicit）属性实现代替数据成员可以极大减少打字的功夫。如果你还是使用 public 变量（variables）在你的类中，请停下来！属性可以暴露你的数据成员像 public 接口（interface）一样，更提供你想要面向对象的封装。属性是语言基本元素，可以和访问数据成员的方式一样访问，但却是用方法实现的。

一个类的有些成员最好表示为数据：客户的名字，一个的坐标（x,y），或者去年的收入。属性是可以让你创建一个表现和数据访问一样又有所有方法的优势的接口。客服端代码可以像访问 public 域（fiedls）访问属性。但是实际是使用方法去定义属性访问的行为。

.NET 框架假定你会使用属性来访问你的 public 数据成员。实际上，在 .NET 框架数据绑定（data binding）类支持属性而不是 public 数据成员。这可以从所有数据绑定类库： WPF， Windows Forms， Web Forms， 和 Silverlight 。数据绑定就是绑定对象（object）到用户界面（user interface）控制上。数据绑定的使用反射（refelction）去查找一个类的已知属性：

```
textBoxCitye.DataBindings.Add("Text",address,"City");
```

上面的代码实现绑定 textBoxCity 的属性 Text 到对象 address 的属性 City 。如果使用 public 数据成员 City 是不能工作的，这些框架类库设计不支持这样的实践。 public 数据成员是一个不好的实践，所以没有被支持。这样的决定只是给出另一个理由——按照正确的面向对象计算取实践。

你也许会说，数据绑定只是应用于那些含有要在用户界面显示的元素的类。但那不意味着属性专门使用在用户界面的逻辑中。你应该使用属性在其他类和结构体（structures）。属性能最快捷地适应因新的需求或行为的改变。你能很快觉得你自定义的类的 name 不能为空。如果你使用 public 属性 Name，你可以很快修复这个问题在一个地方：

```
public class Customer
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
            "Name cannot be blank",
            "Name");
            name = value;
        }
        // More Elided.
    }
}
```

如果你使用 public 数据成员，你会很困难地去找遍你的程序，找到每一处并且修复。这花费了太多时间，太多时间了。

因为属性是用方法实现的，增加多线程的支持是很容易的。你可以加强 get 和 set 访问器（accessors）的实现来提供数据访问的同步。

```
public class Customer
{
    private object syncHandle = new object();
    private string name;
    public string Name
    {
        get
        {
            lock (syncHandle)
            return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
            "Name cannot be blank",
            "Name");
            lock (syncHandle)
            name = value;
        }
    }
    // More Elided.
}
```

属性有所有方法的语言特性。属性可以是 virtual ：

```
public class Customer
{
    public virtual string Name
    {
        get;
        set;
    }
}
```

你也许注意到上一个例子使用 C# 3.0 的隐式属性语法。创建属性包裹一个备份存储（backing store）是一个常见的模式。一般，你不需要验证逻辑在属性的 getters 或 setters 。 C# 语言支持最简化的隐式属性语法来极大减少需要作为属性来暴露简单域的打字功夫。编译器会创建一个 private 成员域（通常被称作备份存储）并且实现 get 和 set 放弃的逻辑。

你可以使用简化的隐式属性拓展属性为 abstract 和定义属性作为接口的一部分。下面的例子演示了在泛型接口（generic interface）定义了属性。可以注意到下面定义的接口没有包含任何实现并且语法和隐式属性是一致的。每个实现这个接口的类都要实现这个规范。

```
public interface INameValuePair<T>
{
    string Name
    {
        get;
    }
    T Value
    {
        get;
        set;
    }
}
```

属性是全面的（full-fledeged），语言第一类元素：扩展方法以访问或修改内部数据。任何成员函数能做的，你都可以使用属性。

属性的两个访问器会被编译成类的两个单独的方法。在你的 C# 程序中可以指定 get 和 set 访问器不同访问修饰符。这让你可以很好控制通过属性暴露数据成员的可访问性。

```
public class Customer
{
    public virtual string Name
    {
        get;
        protected set;
    }
// remaining implementation omitted
}
```

属性的语法扩展远超过简单的数据域。如果你的类包含可索引项（indexed items）作为它的接口，你可以使用索引器（indexers）（参数化属性（parameterized properties））。创建属性实现返回序列（sequence）中的项是非常有用的：

```
public int this[int index]
{
    get { return theValues[index]; }
    set { theValues[index] = value; }
}
// Accessing an indexer:
int val = someObject[i];
```

索引器能和单一项（single-item）属性一样被语言全面支持：索引器可以像你自己写的方法一样实现，里面可以应用你的校验和计算。索引器可以是 virtual 或 abstract， 可以在接口内声明，并且可以是只读或读和写。参数是数字的一维索引器可以参与数据绑定。其他使用非整数参数的索引器用来定义 maps 和 dictionaries ：

```
public Address this[string name]
{
    get { return adressValues[name]; }
    set { adressValues[name] = value; }
}
```

为了和多维数组保持一致，你可以创建多维索引器，在不同轴（axis）使用相同或不同类型：

```
public int this[int x, int y]
{
    get { return ComputeValue(x, y); }
}
public int this[int x, string name]
{
    get { return ComputeValue(x, name); }
}
```

值得注意的是所有索引器必须使用 this 关键字声明。在 C# 中你不能自己命名索引器。所以一个类型的索引器必须有不同的参数列表来避免歧义。几乎所有的属性的功能都适用索引器。索引器可以是 virtual 或 abstract ；索引器的 setters 和 getters 可以不同的访问限制。不过，你不能像创建隐式属性一样创建隐式索引器。

属性的功能都非常好，是一个非常不错的改进。如果你还蠢蠢欲动使用数据成员的初始实现，然后等到你需要使用到属性的优势的时候，再用属性替换数据成员。这听起来像是一个合理的策略——但这是错误的。考虑下面的类的定义：

```
// using public data members, bad practice:
public class Customer
{
    public string Name;
    // remaining implementation omitted
}
```

类 Customer 有一个数据成员 Name 。你可以使用熟悉的成员记号（member notation） get 或 set 这个 Name ：

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

这个简单有直观。你会觉得你以后使用属性替换数据成员 Name ，而且代码不做改动能照常工作。好吧，确实是那样的。属性就是访问起来跟数据成员一样。这个语法的目的就在于此。但是属性不是数据。属性访问和数据访问会产生不同的微软中间语言（Microsoft Intermediate Language）指令。

尽管属性和数据成员在代码上兼容的。但在二进制上是不兼容的。一个很明显的案例，当你把一个 public 数据成员改为等同的 public 属性，意味着你要重新编译所有使用这个 public 数据成员的代码。 C# 把二进制程序集（assemblies）看做“第一类公民”。语言的一个目的就是你可以发布单一更新的程序集，而不要更新整个应用。把数据成员改为属性这么简单的行为却破坏二进制的兼容性。这个行为使得部署更新单一程序集变得更困难。

如果你看了 IL 的属性实现，你可能很想知道属性和数据成员的相对性能。属性并不比数据成员快，但也不会慢多少。 JIT 编译器对属性的访问器做了 inline 的优化。如果 JIT 编译器做了 inline 属性访问器优化， 数据成员和属性的性能是一样的。即使没有属性访问器没有置为 inline，性能差别也只是可以忽略不计的一个函数调用话费。这差别只有在少数情况下才可以被测量出来。

属性是像数据一样被调用的方法。调用者就会有一些访问权限期望。他们把访问属性当做数据访问。毕竟，那就是属性。你的属性访问器要能满足这些期望。 Get 访问器不能有其他作用。 Set 访问器改变了状态，调用者要能看到改变。

属性访问器同时满足使用者的性能期望。属性访问跟数据域访问很类似。这不能导致属性和简单数据访问有着显著不同的性能特点。属性访问不能在长计算，或跨平台调用（比如执行数据库查询），或其他长操作和使用者期望保持一致。

无论什么时候，你要在类中暴露数据作为 public或 protected 接口，请使用属性。对于序列或 dictionaries 是使用索引器。所有数据成员应该无一例外地使用 private 修饰符。你会立即在数据绑定得到支持，而且在以后会很容易方法中的实现。把变量封装在属性里所有的打字功夫加起来就一到两分钟。你会发现在使用属性替换之前的设计会花费数小时。现在花一点时间，以后节约你大量时间。

小结：

万事开头难，终于写下了翻译 Effective C# : 50 Specific Ways to Improve Your C# 2nd Edition 的第一篇，花的时间有点心疼，感觉颈椎都熬出问题了（太专注了）。之前工作都没觉得什么，看来还是工作不要那么专注，以后把时间挤出来干这个，加油！发现最后一句话说的蛮好的：Spend a little time now, and save yourself lots of time later. 分享给大家.....

因为是第一次翻译，欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在 文首 注明出处：http://dsqiu.iteye.com/blog/1976256

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则2：偏爱 readonly 而不是 const

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

C# 有两种常量：编译时常量和运行时常量。它们有不同的行为，不当使用会花费性能或出现错误。如果不得不选择其中一个，宁可是一个慢一点但正确的程序，而不是一个快速但会出错的程序。出于这个考虑，你应该更偏向于运行时常量而不是编译时常量。编译时常量会比编译时常量稍微快点，但更不灵活。只有当性能是一个致命因素而且要求常量不随版本发生改变时才会保留编译时常量。

你可以使用 readonly 关键字声明运行时常量。使用 const 关键字声明编译器常量：

```
// Compile time constant:
public const int Millennium = 2000;
// Runtime constant:
public static readonly int ThisYear = 2004;
```

上面的代码可以出现类或结构体的作用域（scope）中。编译时常量还可以在方法体中声明。运行时常量不能在方法体重声明。

编译时常量和运行时常量访问方式不同导致不同的行为。在目标代码中编译时常量会被替换成常量值。比如下面代码：

```
// Compile time constant:
public const int Millennium = 2000;
// Runtime constant:
public static readonly int ThisYear = 2004;
```

会和下面写法的编译的 IL 代码是一样的：

```
if (myDateTime.Year == 2000)
```

运行时常量的值是在运行时得到的。当你引用一个只读（read-only）常量， IL 会引用一个 readonly 变量而不是直接使用值。

使用编译时常量和运行时常量还有不同的限制。编译时常量只能在基本类型（内建整数和浮点数类型），枚举类型，或字符串。编译时常量要求类能用有意义的常量赋值初始化。而只有基本类型才能在 IL 代码中使用常量（literal values）来替换。不能使用使用 new 操作法初始化编译时常量，即使它是一个值类型：

```
// Does not compile, use readonly instead:
private const DateTime classCreation = new
    DateTime(2000, 1, 1, 0, 0, 0);
```

编译时常量只能使用与数字和字符串。只读（Read-only）变量也是常量，即不能在构造函数完成之后再修改。但只读变量是在运行时赋值。这会比编译时常量更灵活。首先，运行时常量可以是任何类型。你必须在构造函数或者直接初始化。你可以让 DateTime 结构体变为 readonly 值；但不能使用 const 创建 DateTime 值。

readonly 值可以是实例类型（instance）常量，让一个类的实例存储不同值。而编译时常量则是被定义为 static 常量的。

只读变量最重要的不同在于运行时才确定值。当你引用一个只读变量， IL 会为你产生一个指向只读变量的引用，而不是值。这种差异将对维护上产生深远的影响。编译时常量产生的 IL 代码就跟直接使用数值变量时一样的，即使是跨程序集：一个程序集的常量在另一个程序集还是被替换为数值。

编译时常量和运行时常量的赋值方式会影响运行时的兼容性。假设你在程序集 Infrastructure 中同时定义了 const 和 readonly 域：

```
public class UsefulValues
{
    public static readonly int StartValue = 5;
    public const int EndValue = 10;
}
```

在另外一个程序集，你引用这两个值：

```
for (int i = UsefulValues.StartValue;
    i < UsefulValues.EndValue; i++)
    Console.WriteLine("value is {0}", i);
```

如果你运行这个简单的测试程序，很明显你会得到下面的输出：

```
Value is 5
Value is 6
...
Value is 9
```

一段时间后，你发布新版本的 Infrastructure 程序集并作下面的改动：

```
public class UsefulValues
{
    public static readonly int StartValue = 105;
    public const int EndValue = 120;
}
```

你只发布程序集 Infrastructure 而没有重新编译全部应用程序。你希望得到下面的结果：

```
Value is 105
Value is 106
...
Value is 119
```

实际上，你不会得到任何输出。循环条件开始于105，结束于10。 C# 编译器用10替换应用程序集的 const 变量而不是指向存储 EndValue 的引用。 StartValue 的情况趋势截然不同。因为它被声明为 readonly ：在运行时确定值。因此，应用程序集能不用重新编译就能充分利用新值；只要很简单地按照更新版本的 Infrastructure 程序集就可以改变使用该变量的值。更新 public const 变量的值应该当做接口的变化。更新只读常量的值只是实现的改变，兼容客户端的二进制代码。

另一方法，有时候某些值的确需要是编译时常量。例如：考虑使用编译时常量标记对象的序列化版本（查看原则27）。标记特定版本号的持久化值要使用编译时常量，它们决不会发生改变。但当前的版本号应该是一个运行时常量，随着版本不同而改变。

```
private const int Version1_0 = 0x0100;
private const int Version1_1 = 0x0101;
private const int Version1_2 = 0x0102;
// major release:
private const int Version2_0 = 0x0200;
// check for the current version:
private static readonly int CurrentVersion =
    Version2_0;
```

你会使用运行时常量存储每个文件的当前版本号：

```
// Read from persistent storage, check
// stored version against compile-time constant:
protected MyType(SerializationInfo info,
    StreamingContext cntxt)
{
    int storedVersion = info.GetInt32("VERSION");
    switch (storedVersion)
    {
        case Version2_0:
            readVersion2(info, cntxt);
            break;
        case Version1_1:
            readVersion1Dot1(info, cntxt);
            break;
        // etc.
    }
}
```

```
// Write the current version:
[SecurityPermissionAttribute(SecurityAction.Demand,
SerializationFormatter = true)]
void ISerializable.GetObjectData(SerializationInfo inf,
    StreamingContext cxt)
{
    // use runtime constant for current version:
    inf.AddValue("VERSION", CurrentVersion);
    // write remaining elements...
}
```

相比 readonly，使用 const 最后的一个优势就是性能：已知的常量值会比使用变量访问的 readonly 变量产生稍微高效的代码。然而，性能上甚微的收效和灵活性的减小应该做一个很好的权衡。放弃灵活性之前一定要剖析性能差异。可选参数的默认值会在调用时会像编译时变量（声明为 const 的变量）一样被替换成默认值。和使用 readonly 和 const 变量一样，你要非常认真对待可选参数值的不同。（查看原则10。）

当你使用命名（named）参数和可选（optial）参数时，你会遇到和使用运行时常量和编译时常量一样的权衡。

当在编译时期必须要获得变量的值时必须使用 const ：特性（attribute）参数和枚举定义，以及当你定义一个不随版本的变化而变化的值得罕见的时候。无论如何，更偏爱于只读常量的更强的灵活性。

小结：

第二节字数相对少些，所以今天虽然颈椎有点不舒服（千万不要有事呀，我还没有疯够），根据以前的节奏（时间还早），还可以干点别的，或者躲进被窝理顺事情，每天给自己思考的事件太少了，没有思考，积淀就会来的慢，这个跟前面强调的不同哈。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1976703

更多精彩请关注D.S.Qiu的 博客 和微博（ID：静水逐风）

原则3：选择 is 或 as 而不是强制类型转换

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

当你使用 C#，你就应该知道它是强类型语言。大多时候，这总是一个好事。强类型意味着编译器可以发现你的代码中的类型是否匹配。也可以减少运行时的类型检查。但是有些时候，类型检查是不可避免的。很多时候，函数的调用参数是 object，因为在框架中定义好了函数原型。你很有可能要进行强制类型转换（cast）为其他类型的类或接口。你可以用两种选择：使用 as 操作符或者强制类型转换。在转换之前，你需要进行变量类型防护：使用 is 进行类型判断，然后再用 as 进行转换或者强制类型转换。

无论什么时候都应该选择 as 操作符，因为它比盲目的强制类型转换更安全且运行时更高效。as 和 is 不会执行任何用户自定义的转换。只有当目标类型和运行时类型匹配时才会成功转换；它不会构造一个新对象去满足需求。

看一个例子。你需要些一段代码将任意类型的对象转换为 MyType 的实例。你可能会这样写：

```
object o = Factory.GetObject();
// Version one:
MyType t = o as MyType;
if (t != null)
{
// work with t, it's a MyType.
}
else
{
// report the failure.
}
```

或者，你会这样写：

```
object o = Factory.GetObject();
// Version two:
try
{
MyType t;
t = (MyType)o;
// work with T, it's a MyType.
}
catch (InvalidCastException)
{
// report the conversion failure.
}
```

你会觉得第一个版本的写法简单而可读性强。它没有 try/catch 代码块，所以可以同时避免性能开销和代码量。注意到强制类型转换除了要捕捉异常之外还要坚持是否为 null 。null 类型可以强制转换为任意引用类型，但 as 操作符会返回 null 。所以强制类型转换要检查 null 和捕捉

异常。而使用 as，你只要很简单检查返回引用是否为 null 。

强制类型转换和 as 操作符转换最大的区别在于如何看待用户自定义的类型转换。 as 和 is 操作符在运行时要转换对象的类型，不会进行额外的操作。如果一个类型不是转换要求的类型或者是其子类的话，转换就会失败。然而，强制类型转换可以使用转换操作符转换一个对象到需要的类型。这就包括了内建数据类型的转换。强制转换一个 long 整数到 short 整数会丢失信息。

同样的问题也会在你自定义的类型中出现。考虑下面的类型：

```
public class SecondType
{
private MyType _value;
// other details elided
// Conversion operator.
// This converts a SecondType to
// a MyType, see item 9\.
public static implicit operator MyType(SecondType t)
{
     return t._value;
}
}
```

假设， Factory.GetObject() 会返回上一段代码的 SecondType 的对象：

```
object o = Factory.GetObject();
// o is a SecondType:
MyType t = o as MyType; // Fails. o is not MyType
if (t != null)
{
// work with t, it's a MyType.
}
else
{
// report the failure.
}
// Version two:
try
{
MyType t1;
t1 = (MyType)o; // Fails. o is not MyType
// work with t1, it's a MyType.
}
catch (InvalidCastException)
{
// report the conversion failure.
}
```

两种版本的代码都会失败。但是强制类型转换执行的是用户自定义的转换。你应该会觉得会成功的。你是对的——是会成功的，如果按你想的方式。但是还是失败了——因为编译器产生的代码是基于编译时的对象类型， o 。编译器对运行时 o 的类型一无所知； o 只是 object 的对象。编译器没有找到用户定义的从 object 转换为 MyType 的方法。它只是检查了 object 和 MyType 的类型定义。编译器没有发现用户定义的类型转换，会产生在运行时检查 o 的类型是否是 MyType 的代码。因为 o 是 SecondType 对象，所以会失败。编译器不会检查 o 的实际运行时类型是否可以转换为 MyType 对象。

下面代码块可以成功将 SecondType 转换为 MyType ：

```
object o = Factory.GetObject();
// Version three:
SecondType st = o as SecondType;
try
{
MyType t;
t = (MyType)st;
// work with T, it's a MyType.
}
catch (InvalidCastException)
{
// report the failure.
}
```

你应该杜绝写这么糟糕的代码，但这确实解决一个常见的问题。虽然不应该这样写代码，但可以用 object 参数来实现正确的转换：

```
object o = Factory.GetObject();
DoStuffWithObject(o);
private static void DoStuffWithObject(object o)
{
try
{
MyType t;
t = (MyType)o; // Fails. o is not MyType
// work with T, it's a MyType.
}
catch (InvalidCastException)
{
// report the conversion failure.
}
}
```

只需记住用户自定义的类型转换只针对编译时期的类型，而不会对运行时类型起作用。这不会影响 o 运行时类型和 MyType 转换的存在。编译器并不会知道和在意。下面语句会有不同的行为根据 st 不同的声明类型：

```
t = (MyTpe)st
```

下面语句无论 st 声明是什么类型得到的结果都是一样的。所以，应该更偏向使用 as 而不是强制类型转换——它得到的结果更一致。实际上，这次类型没有继承关系，而且用户自定义转换存在，下面的语句会产生一个编译错误：

```
t = st as MyType;
```

既然你知道使用尽可能使用 as，下面我们讨论什么时候是不能使用的。 as 操作符是不能再值类型上使用的。这条语句不能通过编译：

```
object o = Factory.GetValue();
int i = o as int; // Does not compile.
```

这是因为 int 是值类型，不能赋为 null 。那如果 o 不是整数 int 变量 i 会取什么值。无论取什么值都是无效的整数。因此， as 不能使用，只有使用强制类型转换语法。这实际是一个装箱（boxing）或拆箱（unboxing）的转换（查看原则45）。

```
object o = Factory.GetValue();
int i = 0;
try
{
i = (int)o;
}
catch (InvalidCastException)
{
i = 0;
}
```

流控制机制的异常是一个非常糟糕的做法。但你又不得不使用强制类型转换的行为。你可以使用 is 语句去掉可能引起的异常或转换：

```
object o = Factory.GetValue();
int i = 0;
if (o is int)
i = (int)o;
```

如果 o 不是可以转换为 int 的其他类型，比如 double ， 上面的 is 操作会返回 false。参数为 null ， is 总是返回 false 。

is 操作符只应该用于无法使用 as 来转换的情况。否则只是多余：

```
// correct, but redundant:
object o = Factory.GetObject();
MyType t = null;
if (o is MyType)
t = o as MyType;
```

上面的代码和下面写的代码是一样的：

```
// correct, but redundant:
object o = Factory.GetObject();

MyType t = null;
if ((o as MyType)) != null)
t = o as MyType;
```

这是低效并且多余的。如果你打算用 as 转换类型， is 类型检查简单但却没有必要的。检查 as 的返回值是否为 null ，更简单。

既然掌握了 is ， as 和强制类型转换的区别了，那么哪个操作应该在 foreach 循环中使用？

foreach 循环可以操作非泛型 IEnumerable 序列和内建强制类型转换为迭代器（iteration）。（你应该尽可能使用类型安全泛型版本。非泛型版本的存在处于历史目的和支持晚绑定情况）。

```
public void UseCollection(IEnumerable theCollection)
{
foreach (MyType t in theCollection)
t.DoStuff( );
}
```

foreach 使用强制类型转换操作转换对象到循环体中使用的类型。 foreach 语句几乎等同于下面手动实现的版本：

```
public void UseCollectionV2(IEnumerable theCollection)
{
IEnumerator it = theCollection.GetEnumerator();
while (it.MoveNext())
{
MyType t = (MyType)it.Current;
t.DoStuff();
}
}
```

foreach 的强制类型转换需要同时支持 值类型和引用类型。因为使用强制类型转换，foreach 语句呈现的一样的行为，无论目标类型是什么。因为使用了强制类型转换， foreach 循环会引起并抛出 InValidCastException 异常。

你为 IEnumerator.Current 返回的是 System.Object 类型，没有任何转换操作，上面的测试是不合格的。 SecondType 的对象集合不能使用在上面的 UseCollection 函数，因为依你所见强制类型转换会失败。 foreach 语句（使用强制类型转换）不会检查集合中强制类型转换的对象的运行时类型。只是检查 System.Object 类 （ IEnumerator.Current 返回的类型）和声明的循环变量类型（在上面例子中的 MyType ）转换是否可行。

最后，有时你想知道对象的具体类型，而不只是当前类型能不能转换为目的类型。 is 操作符使用在任何继承自目标类型的对象都会返回 ture 。GetType() 方法可以获得对象的运行时类型。它提供了比 is 和 as 更严格的类型测试。 GetType() 返回一个对象的类可以拿来和指定的类型比较。

再次考虑这个函数：

```
public void UseCollectionV3(IEnumerable theCollection)
{
foreach (MyType t in theCollection)
t.DoStuff();
}
```

如果你创建一个继承 MyType 的类 NewType， NewType 的集合可以在 UnseConllection 函数上正确工作：

```
public class NewType : MyType
{
// contents elided.
}
```

如果你想要写一个函数使得所有 MyType 对象都能工作，上面的方法已经可以了。如果你想要这个方法只对 MyType 对象正常工作，你应该使用精确类型比较。这里你可以在 foreach 循环中实现。很多时候，精确的运行时类型对于相等测试是非常重要的（查看原则6）。很多其他比较， as 和 is 提供的 .isinst 指令在语意上是正确的。

.NET 继承类库（BCL）包含一个使用相同类型操作转换序列中元素的方法：Enumerble.Cast<T>() 转换支持实现 IEnumerale 接口的类的序列的每个元素。

```
IEnumerable collection = new List<int>()
{1,2,3,4,5,6,7,8,9,10};

var small = from int item in collection
where item < 5
select item;
var small2 = collection.Cast<int>().Where(item => item < 5).
Select(n => n);
```

上面代码的最后一行的查询产生相同方法调用。这两个例子， Cast<T> 方法都是转换序列中每个元素到目标类型。 Enumerable.Cast<T> 方法使用的是旧的强制类型转换而不是 as 操作符。旧的强制类型转换说明 Cast<T> 不要需要包含类型约束。使用 as 操作符会会有限制，为了不用实现不同的 Cast<T> 方法， BCL 团队选择使用了旧的强制类型转换操作来只产生一个方法。这是你在写代码是需要权衡的。当你需要转换一个泛型参数的对象时，你需要权衡类型约束必要性和强制类型转换的不同行为。

在 C# 4.0，类型系统可以通过使用动态类型或运行时检查来规避。这也是第5章的目的，“ C# 动态编程”。有很多方法可以预期知道对象的行为而不需要知道对象实现的类或接口。你将要学习什么时候该使用这些技术什么时候该避免。

好的面向对象实践告诉我们应该避免使用类型转换，但是有时候却别无选择。如果你不能避免使用类型转换，使用语言提供的 as 和 is 操作符来清晰地表达你的用意。不同方式的强制类型转换有不同的规则。 is 和 as 操作符几乎总是正确的语义，只有当然测试的对象是正确的类型才会成功。选择这些语句来转换类型而不是强制类型转换，因为能返回你期望的成功或失败，而不会有意想不到的影响。

小结：

这篇文字量比较多，今天上班也比较累，状态不好，翻译的感觉也不是很顺，毕竟脑子转的慢了，本来不想写的，晚上下班的时候刚好跟同事聊起强制类型转换和 as 转换。所以才会鼓足劲写完，白天的时候多看看，这篇感觉作者很多细节还是没有将清楚，看了第一版的翻译就该知道——作者写了很多注释，加了自己的理解。由于时间，精力和需求，暂时我就不做具体的代码测试了，好快呀，又到两点了，头也有点晕，所幸的是憋完了，这周的工作内容也很重，加油！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1977594

更多精彩请关注D.S.Qiu的 博客 和微博（ID：静水逐风）

原则4：使用条件特性（conditional attribute）代替 #if

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明 出处 ：**http://dsqiu.iteye.com**

#if/#endif 块在相同的代码会编译成不同的版本，大多数会有调试（debug）和发布（release）两个版本。但我们会很不乐意去使用这个工具。 #if/#endif 块很容易被滥用，写的代码很难理解而且不容易调试。语言设计者应该负责设计能在不同环境产生不同的机器码的工具。 C# 提供了条件特性（conditional attribute），可以根据设置的环境决定函数的调用。使用条件特性会使用 #if/endif 更加清晰。编译器能解析条件特性，所以能很好的确定那段代码被调用。条件特性是应用在函数级别的，所以你应该分开不同的条件代码封装成不同的方法。当你要使用条件代码块的时候，使用条件特性代替 #if/endif 。

很多资深的程序员喜欢使用条件编译来检查对象的先决和后续条件。你会写一个 private 方法去检查所有类型和对象变量。这个方法使用了条件编译以至于只能在调试版本出现。

```
private void CheckStateBad()
{
    // The Old way:
#if DEBUG
    Trace.WriteLine("Entering CheckState for Person");
    // Grab the name of the calling routine:
    string methodName =
        new StackTrace().GetFrame(1).GetMethod().Name;
    Debug.Assert(lastName != null,
        methodName,
        "Last Name cannot be null");
    Debug.Assert(lastName.Length > 0,
        methodName,
        "Last Name cannot be blank");
    Debug.Assert(firstName != null,
        methodName,
        "First Name cannot be null");
    Debug.Assert(firstName.Length > 0,
        methodName,
        "First Name cannot be blank");
    Trace.WriteLine("Exiting CheckState for Person");
#endif
}
```

上面方法使用 #if 和 #endif 编译选项，你会发现在发布版本你实际创建了一个空的方法。CheckState() 会被所有版本中调用，比如发布版本和调试版本。在发布版本没有做任何事情，但你为此付出了函数调用的代价。同时还要有很小花费在加载和 JIT 空程序。

这个实践是正确的，但在发行版本会导致一个微妙的错误。下面这个就是使用条件编译选项的常见错误：

```
public void Func()
{
    string msg = null;
    #if DEBUG
        msg = GetDiagnostics();
    #endif
    Console.WriteLine(msg);
}
```

在调试版本会正确工作，但是你的发行版本只会让你哭笑不得地输出空字符串。当然这不是你想要的。你出了错，编译器就帮不了你了。在条件编译器块中的代码就是你的逻辑。在 #if/endif 块中的代码很难让你诊断出不同版本的不同行为。

C# 有一个更好的选择：条件特性。使用条件特性，能分离出不同函数，只有在特定的环境变量的定义或某些值的设置才会属于你的类。这个功能最常见的好处就是在调试的时候能有可用的声明。 .NET 框架已经提供了基本的通用功能。这个例子告诉我们 .NET 框架的调试功能，已经条件特性是怎么工作的和什么时候添加到你的代码中。

当你创建了 Person 对象，你要写一个方法去检查对象的变量：

```
private void CheckState()
{
    // Grab the name of the calling routine:
    string methodName =
        new StackTrace().GetFrame(1).GetMethod().Name;
    Trace.WriteLine("Entering CheckState for         Person:");
    Trace.Write("\tcalled by ");
    Trace.WriteLine(methodName);
    Debug.Assert(lastName != null,
        methodName,
        "Last Name cannot be null");
    Debug.Assert(lastName.Length > 0,
        methodName,
        "Last Name cannot be blank");
    Debug.Assert(firstName != null,
        methodName,
        "First Name cannot be null");
    Debug.Assert(firstName.Length > 0,
        methodName,
        "First Name cannot be blank");
    Trace.WriteLine("Exiting CheckState for Person");
}
```

我简化了这个方法以至于没有用太多类库的函数。 StackTrace 类使用反射获取正在调用函数的名字。这是消耗性能的，但这简化了工作，比如产生了程序流程的信息。上面代码，检测到调用函数的名字是 CheckState 。如果被 inline 调用会有一个小的风险，另一种方法就是在调用 CheckState 函数的方法使用 MethodBase.GetCurrentMethod() 传入方法名。你很快会明白为什么不使用这个策略。

后面的方法是 System.Diagnositics.Debug 类或 System.Diagnostics.Trace 类的函数。 Debug.Assert 测试条件，且当条件为 false 是程序停止。后面的是条件为 false 是打印出来的信息。 Trace.WriteLine 将诊断信息输出到调试控制台上。所以这个方法实际当 person 对象不正确是输出信息并终止程序。你可以在所有 public 方法或属性中调用这个方法作为先决条件和后续条件：

```
public string LastName
{
get
{
CheckState();
return lastName;
}
set
{
CheckState();
lastName = value;
CheckState();
}
}
```

如果将一个空字符串或 null 赋给 lastName， CheckState 触发一个断言（assert）。然后检验 lastName 的值。这就是你想要做的。

但这额外的检查会在每个例行任务中花费时间。你只是在调试版本中需要额外的检查。那就是为什么会有条件特性：

```
[Conditional("DEBUG")]
private void CheckState()
{
// same code as above
}
```

条件特性告诉 C# 编译器这个方法只能在有 DEBUG 变量的环境中被调用。

条件特性不影响 CheckState 函数代码的产生，修改的是调用者的代码。如果 DEBUG 变量被定义，你的代码是这样的：

```
public string LastName
{
get
{
CheckState();
return lastName;
}
set
{
CheckState();
lastName = value;
CheckState();
}
}
```

如果没有被定义，会是这样的：

```
public string LastName
{
get
{
return lastName;
}
set
{
lastName = value;
}
}
```

无论环境变量状态是怎么样的， CheckState 函数体都是一样的。这就是要告诉我们， .NET 的编译和 JIT 之间的区别。不管 DEBUG 环境环境变量是否定义， CheckState() 方法都会被编译嵌入在程序集中。这可能不是很搞笑，但这只是花费了硬盘的容量。 CheckState() 不会被载入内存和 JITed ，除非被调用。它存在在程序集的二进制文件中是无关紧要的。这个策略增加灵活性而且不消耗性能。阅读 .NET 框架的 Debug 类可以有更加深入的理解。安装了 .NET 框架的机器， System.dll 程序集会有 Debug 类的所有方法代码。当调用者函数被编译，环境变量控制方法是否被调用。运用条件指令可以让你创建的类库嵌入调试特性。这些特性可以运行时打开或关闭。

你可以创建一个方法依赖多个环境变量。当你运用多个条件特性，它们是通过 OR 组合起来的。例如，下面版本的 CheckState 当 DEBUG 或 TRACE 为真时，会调用。

```
[Conditional("DEBUG"),Conditional("TRACE")]
private void CheckState()
```

如果想要使用 AND 构建，你需要使用预处理指令定义预处理符：

```
#if ( VAR1 && VAR2 )
#define BOTH
#endif
```

的确，当你要创建一个依赖多于一个环境变量的条件行为，你不得不退回到之前 #if 的做法。所有 #if 创建一个符号。但要避免在编译选项中加入任何可运行代码。

所以，你可以按下面方式重写 CheckState 方法：

```
private void CheckStateBad()
{
// The Old way:
#if BOTH
Trace.WriteLine("Entering CheckState for Person");
// Grab the name of the calling routine:
string methodName =
new StackTrace().GetFrame(1).GetMethod().Name;
Debug.Assert(lastName != null,
methodName,
"Last Name cannot be null");
Debug.Assert(lastName.Length > 0,
methodName,
"Last Name cannot be blank");
Debug.Assert(firstName != null,
methodName,
"First Name cannot be null");
Debug.Assert(firstName.Length > 0,
methodName,
"First Name cannot be blank");
Trace.WriteLine("Exiting CheckState for Person");
#endif
}
```

条件特性只能运用于整个的方法。除此之外，任何条件特性方法必须是 void 返回值。你在代码块中使用条件特性，也不能创建有返回值的条件特性方法。而是，要分离出具体条件行为的代码单独写成条件特性方法。你应该回顾下那些条件方法对对象状态的副作用，条件特性会比 #if/endif 好很多。使用 #if/endif 块，你会错误的移除了很多重要的方法调用或赋值语句。

前面的例子使用预定义的 DEBUG 或 TRACE 符号。你也可以扩展任何你定义的符号。条件特性可以被多种方式定义的符号控制。你可以定义符号从编译命令行，或从操作系统 shell 的环境变量，或从代码的编译选项中定义。

你应该注意到前面的每个条件特性的方法都是 void 返回值而且没有参数。实际实践应该遵从这个原则。编译器是前置条件特性方法必须是 void 返回值。然后，你可以创建一个方法含有引用类型参数。这种做法会导致副作用，应该尽量避免。考虑下面一段代码：

```
Queue<string> names = new Queue<string>();
names.Enqueue("one");
names.Enqueue("two");
names.Enqueue("three");
string item = string.Empty;
SomeMethod(item = names.Dequeue());
Console.WriteLine(item);
```

SomeMethod 添加了条件特性：

```
[Conditional("DEBUG")]
private static void SomeMethod(string param)
{
}
```

这里会有一个很微妙的错误。 SomeMethod() 只有在 DEBUG 符号被定义了才会被调用。而且 names.Dequeue() 也是一样的。因为结果不是必须的，所以方法没有调用。任何条件特性的方法不应该有任何参数。使用调用方法来产生参数会有副作用。如果条件不为 ture 这些方法不会被调用。

条件特性比 #if/#endif 产生了更高效的 IL 代码。还有一个好处就是只能使用在函数级别上，这迫使你要更好的组织你的条件代码。编译器使用条件特性帮助我们避免了使用 #if/#endif 的常见错误。条件特性比预处理更能让你你的条件代码分离的更清晰。

小结：

更新晚了，昨天晚上写了一半，现在弄完。昨天家里发生了点事情，心里一直不安，感觉挺无奈的。只有对自己说，我要努力，我要顶住。原则4，相对于前面3个原则有点偏门，而且两点少了点，说服力不够。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1979093

更多精彩请关注D.S.Qiu的 博客 和微博（ID：静水逐风）

原则5：总是提供 ToString()

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

在 .NET 环境中 System.Object.ToString() 是最常使用方法之一。你应该为客户端代码的所有类写一个合理的版本。否则，你要强迫每个使用你代码的使用者使用你创建可读的说明的属性。字符串说明很容易被用来向使用者说明对象的信息：在 WPF （Windows Presentation Foundation）框架中， Silverlight 框架中， Web Forms 或控制台输出。字符串说明也能被用来调试。你创建的每个类型都应该重写（override）这个方法。当你创建更复杂的类型，你应该实现更复杂的 IFormattable.ToString() 。必须承认：如果你没有重写这个方法，或者写了得很糟糕，你的客户会强制为你修复这个问题。

System.Object 的版本会返回类型的全限定名字。这个信息很少有用：“ System.Drawing.Rect ”，“ MyNamespace.Point ”，“SomeSample.Size ” 不是我们要向客户展示的信息。这就是你没有重写 ToString() 获得的信息。你一个类只写一次，但是你的客户使用很多次。当你的写类多付出一点工作，会从你使用者那里得到回报的。

我们考虑最简单的需求：重写 System.Object.ToString() 。你创建的类你需要重写 ToString 为了提供最常用的文本说明。下面的自定义的类，提供三个 public 属性：

```
 public class Customer
{
    public string Name
    {
        get;
        set;
    }
    public decimal Revenue
    {
        get;
        set;
    }
    public string ContactPhone
    {
        get;
        set;
    }
    public override string ToString()
    {
        return Name;
    }
}
```

继承的 Object.ToString() 版本会发挥 “ Custormer ”。这个信息对任何都没有作用。即使你使用 ToString() 只是为了调试，它应该比那更复杂些。

你应该重写 ToString() 方法，使之返回最有可能是这个类的文本说明。在 Customer 的例子里，就是 name：

```
public override string ToString()
{
    retrun Name;
}
```

如果你不遵从这个原则的其他建议，按照上面的方法为你的类重写该方法。这会直接节省每个人的时间。当你提供了 Object.ToString() 方法的合理的实现，这个类的对象很容易添加到 WPF 控件， Siverlight 控件， Web Form 控件，或打印输出。 .NET BCL 使用在每个控件中使用重写的 Object.ToString() 来说明对象： combo boxes ， list boxes ， text boxes ， 以及其他控件。在 WPF 或 Web Form 中，如果你创建 Customer 的对象列表，你可以通过 System.Console.WriteLine() 来输出System.String.Format() 或 ToString() 的 name 文本。

任何时候 .NET BCL 想要获取一个 Customer 的字符串说明，你的 Customer 类型要提供属性 Name 。上面简单的三行代码处理所有基本的需求。

C# 3.0 编译为匿名类提供了默认的 ToString() 。默认生成的 ToString() 会输出每个元素的属性值。属性的说明的序列是 LINQ 查询结果，会显示他们的类型信息而不是他们的值。看下面这段代码：

```
 int[] list = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var test = new { Name = "Me",
    Numbers = from l in list select l };
Console.WriteLine(test);
```

将会显示：

```
{ Name = Me, Numbers = System.Linq.Enumerable+WhereSelectArrayIterator`2 [System.Int32,Sy
```

甚至，编译器为匿名类产生的输出会比你自定义的类型更好，除非你重写了 ToString() 方法。你应该为使用者提供比编译器在方法作用域中为暂时类型做的更好支持。

之前定义的 Customer 类型有三个属性： Name ， Revenue 和 Phone。 重写的 System.ToString() 只使用了 Name 。你可以通过实现 IFormattalbe 接口来解决这个缺陷。IFormattable 包含了一个重载的 ToString() 方法让你指定类的格式化输出信息。当你需要创建不同格式的字符串输出是，你就需要这个接口。定义的类型是这个接口的对象。使用可以创建一个报表，以表格形式输出包含 Customer 的 Name 和去年的收入 Revenue 。IFormattalbe.ToString 就是提供这个作用，让你能格式化类的输出字符串。IFormattable.ToString() 方法声明（signature）包含了一个要格式化的字符串和格式化提供器：

```
string System.IFormattable.ToString(string format, IFormatProvider formatProvider)
```

你可以使用格式化字符串来指定类型自己的格式。你可以指定格式化字符的关键字符。在这个 Customer 的例子里，你可以指定 n 表示 Name， r 表示 Revenue， p 表示 Phone 。同时还可以向下面 IFormattable.ToString() 版本一样，指定组合关键字符：

```
// supported formats:
// substitute n for name.
// substitute r for revenue
// substitute p for contact phone.
// Combos are supported: nr, np, npr, etc
// "G" is general.
string System.IFormattable.ToString(string format, IFormatProvider formatProvider)
{
if (formatProvider != null)
{
ICustomFormatter fmt = formatProvider.GetFormat(
this.GetType()) as ICustomFormatter;
if (fmt != null)
return fmt.Format(format, this, formatProvider);
}
switch (format)
{
case "r":
return Revenue.ToString();
case "p":
return ContactPhone;
case "nr":
return string.Format("{0,20}, {1,10:C}",
Name, Revenue);
case "np":
return string.Format("{0,20}, {1,15}",
Name, ContactPhone);
case "pr":
return string.Format("{0,15}, {1,10:C}",
ContactPhone, Revenue);
case "pn":
return string.Format("{0,15}, {1,20}",
ContactPhone, Name);
case "rn":
return string.Format("{0,10:C}, {1,20}",
Revenue, Name);
case "rp":
return string.Format("{0,10:C}, {1,20}",
Revenue, ContactPhone);
case "nrp":
return string.Format("{0,20}, {1,10:C}, {2,15}",
Name, Revenue, ContactPhone);
case "npr":
return string.Format("{0,20}, {1,15}, {2,10:C}",
Name, ContactPhone, Revenue);
case "pnr":
return string.Format("{0,15}, {1,20}, {2,10:C}",
ContactPhone, Name, Revenue);
case "prn":
return string.Format("{0,15}, {1,10:C}, {2,15}",
ContactPhone, Revenue, Name);
case "rpn":
return string.Format("{0,10:C}, {1,15}, {2,20}",
Revenue, ContactPhone, Name);
case "rnp":
return string.Format("{0,10:C}, {1,20}, {2,15}",
Revenue, Name, ContactPhone);
case "n":
case "G":
default:
return Name;
}
}
```

添加下面函数，让你客户代码能够指定 Customer 数据的显示：

```
IFormattable c1 = new Customer();
Console.WriteLine("Customer record: {0}",c1.ToString("nrp", null));
```

针对不同的类，IFomrattable.ToString() 的实现特定的，但当你实现 IFormattable 接口，你必须实现某些情况。首先，你必须支持一般格式 “G”。第二，你要支持空格式："" 和 null 。你重写的 Ojbect.ToString() 方法必须返回为这三个格式返回相同的字符串。 .NET 记录类库（BCL）为每个实现了 IFormattable 的类调用 IFormattable.ToString() 而不是 Object.ToString() 。.NET BCL 经常使用格式字符串 null 来条用 IFormattable.ToString()，但很少使用格式字符 G 来格式化字符串。如果添加了 IFormattable 接口去没有支持这个三个标准格式，你会打破 BCL 的字符串自动转换的规范。你会发现这个的 IFormattable 很快会失控。你不可能预料到你的类型需要支持所有可能的格式。大多时候，选择少数常见的格式。客户端代码应该弥补所有边界情况。

IFormattable.ToString() 的第二个参数需要是实现 IFormatProvider 接口的对象。这个对象可以提供你没有预料的到格式的支持。如果你浏览了前面 IFormattable.ToString() 的实现，毫无疑问你会发现数字格式的选项是没有提供的。提供数字的输出是理所当然德尔。无论你支持了多少种格式字符，使用者总有一天会发现一些格式是没有预料到的。那就是为什么方法中头几行去查看实现 IFormatProvider 的对象并委托操作给它定义的 ICustomFormatter 。

把角色从类的定义转移到类的使用者上去。你会发现你需要的格式没有被支持。举个例子，你的 Customer 的 Name的字符长度大于20，你需要修改了格式为 Customer Name提供50个字长度的格式输出。

那就是为什么IFormatProvider 接口的产生。你创建的类可以通过实现 IFormatProvider 和一个实现 ICustomerProvider 的组合类来实现自定义的格式输出。 IFormatProvider 接口定义了一个方法： GetFormat() 。 GetFormat() 方法返回了实现了 ICustomFormatter 接口的对象。ICustomFormatter 接口指定了实际的输出格式的方法。下面两个类的创建，修改使用50列来输出 Customer Name 的输出：

```
// Example IFormatProvider:
public class CustomFormatter : IFormatProvider
{
#region IFormatProvider Members
// IFormatProvider contains one method.
// This method returns an object that
// formats using the requested interface.
// Typically, only the ICustomFormatter
// is implemented
public object GetFormat(Type formatType)
{
if (formatType == typeof(ICustomFormatte))
return new CustomerFormatProvider();
return null;
}
#endregion
// Nested class to provide the
// custom formatting for the Customer class.
private class CustomerFormatProvider :ICustomFormatter
{
#region ICustomFormatter Members
public string Format(string format, object arg, IFormatProvider formatProvider)
{
Customer c = arg as Customer;
if (c == null)
return arg.ToString();
return string.Format("{0,50}, {1,15}, {2,10:C}",c.Name, c.ContactPhone, c.Revenue);
}
#endregion
}
}
```

GetForm 方法返回创建实现 ICustomFormatter 接口的对象。在所请求的方式中，ICustomFormatter.Format() 做的实际格式输出工作。你可以使用 ICustomFormatter.Format() 定义格式字符串，满足在一个历程中指定多个格式的需求。 格式提供器是从 GetFormat 方法返回的 ICustomFormatter 。

为了指定你定义的格式，你需要显示调用参数为 IFormatProvider 对象的 string.Format()：

```
Console.WriteLine(string.Format(new CustomFormatter(), "", c1));
```

现在把角色切回到类的定义者。重写 Object.ToString() 是一个非常简单的提供类的字符串说明的方式。当你定义一个类的时候，都应该重写这个方法。它是最明显和常用的类的说明。它不会太冗余。它会出现在控件，HTML 页，或其他人可读的地方。当罕见的类需要更复杂的输出信息的时候，你就可以充分利用 IFormattable 接口的实现。这可以提供标准方式去定义类的文本输出。如果你对此置若罔闻，你就需要自己实现的格式（formatter）。 这个解决方案需要更多的代码，因为使用者不会检查对象的内部状态。同样，实现这不可能预料所有潜在的格式。

最后，使用者会猜测你的类型的信息。会理解文本输出，所以你需要尽可能提供最简单的信息：所有的类都重写 ToString() 。使得 ToString() 输出的简短而合理。

小结：

这几天工作比较忙，就每天腾出体力翻译了。这篇跟上篇比较类似，记住一个标题就可以了，更多是是一种编程思维，自从工作了，就发现自己太弱了， 以前读书，觉得父母赚钱难，现在自己工作了，这是多么痛的领悟呀......，要加油，不要放弃，我也会有《放牛班的春天》的。竟然上了台，就要相信自己可以做到——《激战》。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1979985

更多精彩请关注D.S.Qiu的 博客 和微博（ID：静水逐风）

原则6：理解几个不同相等概念的关系

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明 出处 ：**http://dsqiu.iteye.com**

当你定义类型（类或结构体）时，你同时要定义类型的相等。 C# 提供四种不同的函数决定两个不同对象是否“相等”：

```
public static bool ReferenceEquals (object left, object right);
public static bool Equals (object left, object right);
public virtual bool Equals(object right);
public static bool operator ==(MyClass left, MyClass right);
```

C# 语言运行你实现这四个函数的自己的版本。但不意味着你需要这么做。你不需要重定义前面两个静态函数。你经常会创建你自己实例方法 Equals() 去定义你定义类型的语义，有时也会重写操作符==() ，尤其是值类型。此外，这四个函数是有关联的，所以你改变其中一个，可能会影响其他函数的行为。所以你需要完全测试这个四个函数。但是不担心，你可以简化它。

当然，这四个方法不是判断是否相等的唯一选择。还可以通过类型去实现 IEquatable<T> 重写 Equals() 。如果类型是值类型的需要实现 IStructuralEquality 接口。这就说，总共有6中不同的方法去表达相等。

和 C# 中复杂的元素一样，这个也遵守这个事实 ： C# 运行你同时创建值类型和引用类型。两个引用类型变量当它们引用相同的对象时相等。好像引用它们的ID一样。两个值类型的变量当它们是相同的类型而且包含相同的内容时才相等。这就是为什么对这些方法都进行相等测试。

我们先从两个不会修改的方法开始。 Object.ReferenceEquals() 当两个变量指向相同对象——也就是说，两个变量含有相同对象的ID时返回 true 。是否比较值类型或引用类型，这个方法总是测试对象ID，而不是对象的内容。也就是说，当你使用测试两个值类型变量相等时， ReferenceEquals() 会返回 false 。即使你拿一个值类型变量和自己比较，ReferenceEquals() 也会返回 fasle 。这是因为封箱操作，你可以在原则45找到相关内容。

```
 int i = 5;
int j = 5;
if (Object.ReferenceEquals(i, j))
    Console.WriteLine("Never happens.");
else
    Console.WriteLine("Always happens.");
if (Object.ReferenceEquals(i, i))
    Console.WriteLine("Never happens.");
else
    Console.WriteLine("Always happens.");
```

你绝不需要重定义 Object.ReferenceEquals()，因为它已经支持了它的功能了：测试两个不同变量的对象ID。

第二个你不要重新定义的是静态方法 Object.Equals()。当你不知道两个参数的运行时参数是，用这个方法可以测试两个变量是否相等。记住 System.Object 是 C# 所有类型的最终基类。无论什么时候，你比较的两个变量都是 System.Object 的实例。值类型和引用类型都是 System.Object 的实例。来看下当不知道类型是，这个方法是如何判断两个变量是否相等的，相等是否依赖类型？答案很简单：这个方法即使把职责委托给其中一个正在比较的类。静态 Object.Equals() 方法的是像下面这样实现的：

```
public static new bool Equals(object left, object right)
{
    // Check object identity
    if (Object.ReferenceEquals(left, right) )
        return true;
    // both null references handled above
    if (Object.ReferenceEquals(left, null) || Object.ReferenceEquals(right, null))
        return false;
    return left.Equals(right);
}
```

上面实例代码引入一个还没有介绍的方法：即，实例的 Equals() 方法。我将会详细介绍，但是我还没打算终止对静态 Equals() 的讨论。我希望你能明白静态 Equals() 方法使用了左参数的实例 Equals() 方法来判断两个对象是否相等。

和 ReferenceEquals() 一样，你不需要重载或重定义自己版本的静态 Object.Equals() 方法因为它已经做了它需要做的事情：当我们不知道运行时类型时，决定两个对象是否相等。因为静态 Equals() 把比较委托给左边参数的实例 Equals() 方法，就是利用这个规则来处理类型的。

既然你明白了为什么不需要重定义静态 ReferenceEquals() 和静态 Equals() 方法。接下来就讨论下你需要重写的方法。但是首先，让我们简要来讨论相等的关系的数学特性。你需要保证你定义的和实现的方法要和其他程序员的期望是一致的。这几意味着你需要关心数学的相等关系：相等是自反的，对称的，可传递的。自反性就是说任何对象都和自身相等。无论类型是什么 a == a 总是 true 的。对称型即与比较的次序是没有关系的：如果 a == b 是 true，b == a 同样也是 true 的。如果 a == b 是 false， b == a 也是 false 。最后一个性质就是如果 a == b 而且 b == c 都是 ture，那么 a == c 必须是 true 的。这就是传递性。

现在是时候讨论实例的 Object.Equals() 函数了，包括什么时候和怎么样重写它。当你发现默认的 Equals() 的行为和你的类型不一致时，你就需要创建自己的实例 Equals() 版本。Ojbect.Equals() 方法使用对象的ID来决定两个变量是否相等。默认的 Object.Equals() 函数和 Object.ReferenceEquals() 的表现是一样的。等等——值类型是不同的， System.ValueType 没有重写 Object.Equals()。记住 ValueType 是所有值类型（使用 struct 关键字）的基类。两个值类型变量当它们类型相同和有相同的内容时是相等的。 ValueType.Equals() 实现就是这个行为。不好的是， ValueType.Equals() 没有一个很高效的实现。 ValueType.Equals 是所有值类型的基类。为了提供正确的行为，你必须在不知道对象的运行时类型的情况下比较子类

的所有成员变量。在 C#，会使用反射来做。你可以查看下原则43。反射有很多不足的地方，尤其当性能是目标时。 相等是在程序中会被频繁调用的集成操作之一，所以性能是值得考虑的。在大多数情况下，你可以重写一个更快的值类型 Equals() 。对于值类型的建议是很简单的：当你创建一个值类型，总是重写 ValueType.Equals() 。

只有当你想要定义引用类型的语义是，你需要重写实例 Equals() 函数。 .NET 框架的一些类都是使用值类型而不是引用类型来判断是否相等。两个 string 对象相等当它们的内容是一样的。两个 DataRowView 对象相等当它们指向同一 DataRow 。关键就是要你的类型服从值语义（比较内容）而不是引用语义（比较对象的ID），你应该重写你自己的实例 Equals() 。

既然你知道什么时候去重写你自己的 Object.Equals() ，你需要命名怎么样实现它。值类型的相等关系封箱有很多补充，在原则45会被讨论。对于引用类型，你的实例方法需要保留之前的行为，避免给使用者惊讶。当你重写 Equals() ，你的类型要实现 IEquatable<T> 。对这点，我会解释的更多一点。这里标准模式只是重写了 System.Object.Equals 。高亮的代码是改为实现 IEquatable<T>。

```
public class Foo : IEquatable<Foo>
{
    public override bool Equals(object right)
    {
        // check null:
        // this pointer is never null in C# methods.
        if (object.ReferenceEquals(right, null))
            return false;
        if (object.ReferenceEquals(this, right))
            return true;
        // Discussed below.
        if (this.GetType() != right.GetType())
            return false;
        // Compare this type's contents here:
        return this.Equals(right as Foo);
    }
    #region IEquatable<Foo> Members
    public bool Equals(Foo other)
    {
        // elided.
        return true;
    }
    #endregion
}
```

首先， Equals() 不能抛出异常——这个没有任何意义。两个变量比较只有相等和不相等，没有其他结果。像 null 引用或错误参数类型的所有错误情况都应该返回 false 。现在，我们详细分析下这个方法的代码，命名为什么每一步的检查和哪些检查是可以被遗漏的。第一个检查右边蚕食是否为 null 。没有任何在 this 的引用上没有任何检查。 C# 中，它是一定不会为 null 的。 CLR 在通过 null 引用调用实例方法会抛出异常。下一个检查是否两个对象的引用是否一样，测试两个对象的ID。这是一个非常有效的测试，内容要相同对象的ID一定要相同。

在下一个检查要比较的两个对象是否是同一个类型。正确的形式是非常重要的。首先，主要是不是假定这就是 类 Foo ；而是调用 this.GetType() 。实际类可能是 Foo 的子类。第二，代码检查被比较对象的真正类型。这还不足以保证你可以把右边参数转换为当前类型。这个测试会导致两个微妙的错。考虑下面有关继承结构的例子：

```
public class B : IEquatable<B>
{
    public override bool Equals(object right)
    {
        // check null:
        if (object.ReferenceEquals(right, null))
            return false;
        // Check reference equality:
        if (object.ReferenceEquals(this, right))
            return true;
        // Problems here, discussed below.
        B rightAsB = right as B;
        if (rightAsB == null)
            return false;
        return this.Equals(rightAsB);
    }
    #region IEquatable<B> Members
    public bool Equals(B other)
    {
        // elided
        return true;
    }
    #endregion
}
public class D : B, IEquatable<D>
{
    // etc.
    public override bool Equals(object right)
    {
        // check null:
        if (object.ReferenceEquals(right, null))
            return false;
        if (object.ReferenceEquals(this, right))
            return true;
        // Problems here.
        D rightAsD = right as D;
        if (rightAsD == null)
            return false;
        if (base.Equals(rightAsD) == false)
            return false;
        return this.Equals(rightAsD);
    }
    #region IEquatable<D> Members
    public bool Equals(D other)
    {
        // elided.
        return true; // or false, based on test
    }
    #endregion
}
//Test:
B baseObject = new B();
D derivedObject = new D();
// Comparison 1\.
if (baseObject.Equals(derivedObject))
    Console.WriteLine("Equals");
else
    Console.WriteLine("Not Equal");
// Comparison 2\.
if (derivedObject.Equals(baseObject))
    Console.WriteLine("Equals");
else
    Console.WriteLine("Not Equal");
```

在任何可能的情况下，你都希望看到相等或不想的打印两次。因为一些错误，这已经不是前面的代码了。第二个比较不会返回 true 。基类 B 的对象，不会被转换为 D 的对象。然后第一个比较可能会评估为 true 。子类 D 可以被隐式转换为 B 。如果右边参数的 B 成员可以匹配

左边参数的 B 成员， B.Equals() 认为两个对象是相等的。即使两个对象是不同的类型，你的方法还是认为他们是想的的。这就违背了相等的对称性。这是因为在类的继承结构中自动转换的发生。

如果这样写：把类型 D 对象显式转换为 B ：

```
baseObject.Equals(derived);
```

derivedObject.Equals() 方法总是返回 false 。如果你不精确检查对象的类型，你会很容易进入这种情况，比较对象的次序会成为一个问题。

上面所有的例子中，重写 Equals()，还有另外一种方法。重写 Equals() 意味着你的类型应该实现 IEquatable<T> 。IEquatable<T> 包含一个方法： Equals(T other) 。实现 IEquatable<T> 意味着你的类型要支持一个类型安全的相等比较。如果你认为 Equals() 只有左右两个参数的类型都相等才返回 true 。 IEquatable<T> 很简单地让编译器捕捉多次两个对象不相等。

还有另外一种方法重写 Equals() 。只有当基类的版本不是 System.Object 或 System.ValueType，你就应该调用基类的方法。前面的例子，类 D调用 Equals() 就是在基类B中定义的。然而，类 B 调用的不是 baseObject.Equals() 。System.Object 的版本只有当两个参数引用同一个对象才会返回 true 。这并不是你想要的，或者你应该没有在第一个基类中重写自己的方法。

原则就是这样，如果你创建一个值类型你就要重写 Equals()，如果是引用类型你不想遵循 System.Object 的引用语义就要重写 Equals() 。当你重写你自己的 Equals()，你应该遵循上面列出的要点实现。重写 Equals() 意味着你要重写 GetHashCode() 查看原则7。

我们几乎完成了本原则。 操作符 ==() 是简单的。无论什么时候你创建一个值类型，重定义操作符 ==() 。原因和实例 Equals 函数一样。默认的版本使用反射区比较两个值类型的内容。这是比任何你实现的都更低效的，所以你要自己重写。遵循原则46的建议避免封箱当比较两个值类型时。

注意的是我们没有说当你重写实例Equals() 你就应该重写操作符 ==() 。我说的是你应该重写操作符 ==() 当你创建值类型的时候。你几乎不用重写操作符 ==() 当你创建引用类型是。.NET 框架期望操作符 ==() 所有引用类型遵循引用语义。

最后，我们说下 IStructuralEquality， System.Array 和 Tuple<> 泛型实现这个接口。它让这些类型实现值语义而不强制比较时的值类型。你会对创建一个值类型是否实现 IStructuralEquality 留有疑惑。这个只有在创建轻量级类型需要。实现 IStructuralEquality 声明一个类是可以被组合成一个基于值语义的大对象。

C# 提供了很多方式去测试是否相等。但是你需要考虑提供自定义的它们其中的两种，支持类似的接口。你不需要重写静态 Object.ReferenceEquals() 和静态Object.Equals() 因为它们能提供正确的参数，尽管不知道它们的运行时类。你总是要重写实例 Equals() 和操作符 ==() 对

于值类型可以提供性能。当你想要引用类型相等而不是对象ID相等，你需要重写实例 Equals()。当你重写 Equals()，你就应该是实现 IEquatable<T>。很简单，是不是？

小结：

这篇的内容是我们最熟悉的，翻译的有点匆忙，平时虽然用的很多，但是都没有深入研究过，还是有很多细节很受用的，比如值类型效率问题等。一个星期翻译六篇，从量上看，还是比较满意的，虽然每天都折腾的很晚，今天虽然是周末，我也没有出去过，不过今天的效率太高了。但质还是不尽如人意的，至少没有一点原创的感觉，以后多做回顾和修改。

下周要好好工作，把手上的工作完成到100%，无可挑剔，加油，我可以做得到的！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1980083

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则7：明白 GetHashCode() 的陷阱

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

这是这本书介绍的唯一一个原则专门讲一个你应该尽量避免的函数。 GetHashCode() 只用在一个地方：定义基于哈希 key 集合，典型地， HashSet<T> 或者 Dictionary<K,V> 容器。值得称赞的是很多问题会随着实现基类的 GetHashCode() 而解决。对于引用类型，是能起作用但是低效。对于值类型，基类版本的总是不正确的。所以这很糟糕。写一个既高效又正确的 GetHashCode() 是完全可能的。没有哪个函数会像 GetHashCode() 一样产生这么多讨论和困惑。看完下面的消除所有困惑。

如果你定义的类型没有被用作容器的 key 这就没关系了。Window 控件 Web page 控件，或数据库连接很少被用作容器的 key 。在这些例子中，你什么也不用做。所有引用类型的哈希码是正确的，即使这不是低效的。值类型应该是不可变的（immutable），才是正确的，尽管同样地不高效。你创建的大多数类，最好避免 GetHashCode() 的存在。

如果有一天，你创建的类要被用作哈希 key，你就需要实现 GetHashCode()，所以继续看下面的。基于哈希码的容器可以优化查找。每个对象产出的整数被称为哈希码。对象根据哈希码以桶的形式存储。为了查找某个对象，你请求哈希码而只是在那个桶里查找。在 .NET 中，每个对象都有一个产生自 System.Object.GetHashCode() 的哈希码。任何重写 GetHashCode() 必须遵循下面三个条件：

1.如果两个对象相等（被操作符 == 定义），你产生相同的哈希值。否则哈希码不能用来查找容器中的对象。

2.对任何对象 A， A.GetHashCode() 必须是一个实例不变量。无论什么方法调用 A.GetHashCode() 必须返回相同的值。这保证了一个物体总是存储在正确的桶中。

3.哈希函数应该针对所有输入情况产生一个整数随机分布。这就是为什么基于哈希容器高效。

写一个正确且有效的哈希函数需要掌握大量类的知识来保证遵循以上3条规则。System.Object 和 System.ValueType 定义的版本不具备这个优势。这两个版本是在对你定义的类不知情的情况下提供的默认行为。 Object.GetHashCode() 使用 System.Object 内部域来产生哈希值。每个对象产生的哈希值要保证唯一，而且存储为整数。这些 key 从1开始而且每创建一个对象就递增。对象ID域在 System.Object 构造器中设置，而且不会被后面改动。Object.GetHashCode() 返回对象的哈希值。

基于上面三条规则逐一检查 Object.GetHashCode()。如果两个对象是相等的，Object.GetHashCode() 返回相同的哈希值，除非你重写操作符 == 。System.Object 的操作符 ==() 测试两个对象的ID。 GetHashCode() 返回对象的内部ID域。它就是这样获得的。如

果你重写了自己版本的操作符 == ，你必须同时重写 GetHashCode() 保证符合第一条规则。有关相等的细节查看原则6。

第二条规则是这样的：对象呗创建之后，它的哈希值不会被改变。

第三条规则，对于所有输入情况应该产生一个所有正式的随机分布，而不持有。一个数字序列不是所有整数的随机分布除非你创建了很大数量的对象。通过 Object.GetHashCode() 产生的哈希码都集中在小范围的整数段。

这意味着 Object.GetHashCode() 是正确但不高效。如果你定义基于哈希表的引用类型，System.Object 的默认行为是正确的但慢。如果你创建一个被用做哈希 key 的类，你应该重写 GetHashCode()，让你的类型有一个更好的哈希值分布。

在介绍如果重写 GetHashCode 之前，这部分讨论 ValueType.GetHashCode() 遵循的上面三个规则。 System.ValueType 重写了 GetHashCode()，提供了所有值类型的默认行为。这个版本的哈希码从类型的第一个域返回。考虑下面的例子：

```
public struct MyStruct
{
    private string msg;
    private int id;
    private DateTime epoch;
}
```

MyStruct 对象的哈希码产生自 msg 域。假设 msg 不为 null，下面这段代码总是返回 true ：

```
 MyStruct s = new MyStruct();
s.SetMessage("Hello");
return s.GetHashCode() == s.GetMessage().GetHashCode();
```

第一个规则说的是两个对象相等（操作符 ==() 定义的）必须有相同的哈希码。大多数情况下，这条规则是被值类型遵循的，但是也可以打破它，就跟引用类型一样。ValueType.operator==() 比较 struct 的第一个域，以及其他域。如果要遵从第一条规则，只要重写 operator== 使用第一个域，就能正确工作。任何 struct 的第一个域不过不参与类型的相等比较就违反此规则，打破了 GetHashCode()。

第二条规则规定哈希码必须是实例不变量。只有 struct 的第一个域是不可变的菜遵循这个规则。如果第一个域的值会变，那么哈希值也会改变。那就打破了这条规则。是的，GetHashCode 是任何第一个域在生命期被改变的 struct 破坏了。这就是另外一个原因不可变值类型是你最好的考虑。

第三条规则依赖于类型第一个类的使用。如果第一个域产生一个整数随机分布，而且第一个域也是分布在 struct 的所有值，然而 struct 产生一个均匀分布。然而，如果第一个域有相同的值，这个规则会被违反。小改动之前的 struct ：

public struct MyStruct

{

private DateTime epoch;

private string msg;

private int id;

}

epoch 域被设置为当前日期（不包括时间），所有产生于同一日期的 MyStruct 对象都具体相同的哈希码。这就不能形成均匀的哈希码分布了。

总结下默认行为，Object.GetHashCode() 对于引用类型能正确工作，即使没有产生有效的分布。（如果你重写了 Object.operator==()，你会破坏 GetHashCode()。ValueType.GetHashCode() 只有在 struct 的第一个域是只读的才能正确工作。ValueType.GetHashCode() 产生一个高效的哈希码只有 struct 的第一个域是一个有意义的子集。

如果你想要构建一个更好的哈希码，你需要对你的类型做些约束。理想情况下，你应该创建一个不可变值类型。不可变值类型的 GetHashCode() 的规则会比不做约束类型更简单。现在自己构建一个 GetHashCode() 的实现，并对三条规则进行检查。

首先，如果两个对象相等， operator==() 定义的，它们必须返回相同的哈希值。任何用于产生哈希码的属性或数据都必须在相等判断中同样被考虑。明显，这意味着属性在相等判断和哈希值产生中都被用到。在相等判断用到的属性可能在哈希码计算中没有用到。System.ValueType 的默认行为就是那样做的，这经常意味着规则3会被违反。相同的数据元素应该被用在两者的计算中。

第二条规则是 GetHashCode() 的返回值必须是实例不变量。想象你定义的一个引用类型，Customer ：

```
 public class Customer
{
    private string name;
    private decimal revenue;
    public Customer(string name)
    {
        this.name = name;
    }
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public override int GetHashCode()
    {
        return name.GetHashCode();
    }
}
```

执行下面代码：

```
Customer c1 = new Customer("Acme Products");
myHashMap.Add(c1, orders);
// Oops, the name is wrong:
c1.Name = "Acme Software";
```

c1 有时在 HashMap 中会丢失。当你往 map 加入 c1，哈希码产生自字符串“Acme Products”。当你改变 Customer 的 name 为 “Acme Software”。哈希码改变了。哈希码产生自新 name : “Acme Software”。 c1 存储在定义为 “Acme Products”的桶中，但是它应该是在“Acme Software”中。你就丢失了 Customer 在你自己的集合中。丢失是因为哈希码不是对象的不变量。你在存储对象后改变了正确的桶。

前面情况只有 Customer 是引用类型才会发生。值类型表现不同，但同样会出现问题。如果 Customer 是值类型， HashMap 存储的是 c1 的复制。最后一行改变 name 的值，并不会对 HashMap 存储有任何影响。因为封箱和拆箱和复制一样，在加入集合之后，你不能改变值类型成员的值。

唯一能解决规则2的方法是使用对象的一个或多个不可变属性定义哈希函数。 System.Object 使用不会改变的对象ID来恪守这条规则。 System.ValueType 希望你的类型的第一个域是不会改变的。你不能比让你的变量不可变做的更好了。当你打算将值类型用作哈希集合的 key，它必须是不可变类型。如果你违反了这条建议，将会破坏哈希表。回顾 Customer 类，你可以更改 name 为不可变。高亮表示 Customer 的改变：

```
public class Customer
{
    private string name;
    private decimal revenue;
    public Customer(string name)
    {
        this.name = name;
    }
    public string Name
    {
        get { return name; }
        // Name is readonly
    }
    public decimal Revenue
    {
        get { return revenue; }
        set { revenue = value; }
    }
    public override int GetHashCode()
    {
        return name.GetHashCode();
    }
    public Customer ChangeName(string newName)
    {
        return new Customer(newName) { Revenue = revenue };
    }
}
```

ChangeName 创建一个新的 Customer 对象，使用构造器和对象初始化语法设置当前 revenue。 name 不可变改变修改 Customer 对象 name 的方式：

```
Customer c1 = new Customer("Acme Products");
myDictionary.Add(c1, orders);
// Oops, the name is wrong:
Customer c2 = c1.ChangeName("Acme Software");
Order o = myDictionary[c1];
myDictionary.Remove(c1);
myDictionary.Add(c2, o);
```

你不得不移除原来的 Customer，改变 name， 然后把新的 Customer 对象加入到 Dictionary 中。这看起来比第一个版本更繁琐，但是这个能正确工作。前一个版本运行程序员写不正确的代码。随着强制用于计算哈希值的属性为不可变，增强了代码的正确性。使用你的类型不会出错。是的，这个版本更好的工作。你要强迫开发者写更多的代码，因为这是正确代码的唯一的方法。确保计算哈希值的数据成员要不可变。

第三条规则说的是 GetHashCode() 应该产生所有输入情况的随机分布。满足这个需求特定依赖你创建的类型。如果魔术公式存在，它会被实现在 System.Object， 而且这个原则也不会存在。一个常用而且成功的算法是使用对所有域使用 XOR 返回 GetHashCode() 的值。如果你的类型包含了一些不可变域，使用它们来计算。

GetHashCode() 有一个非常特别的需求：相等的对象必须产生相等的哈希码，而且这些哈希值必须是实例不可变的而且产生一个均匀分布会更有高效。只有不可变类型，三条才会满足。对于其他类型，依靠默认行为，但是明白这里面的陷阱。

小结：

这篇的内容是我们最简单了，好像也没有我们什么事情，只要记住最后一句话： 对于其他类型，依靠默认行为，但是明白这里面的陷阱。本来不想写的，听了某人的鼓励，还是决定坚持，虽然写到了凌晨3点，但还是完成了。所以有时候坚持只是需要一个理由，虽然不太会受这个影响，但是还是会给自己一个心理暗示。

天亮了，还要上班，还要帮国外的帮，也就不能睡觉了，加油，我可以的！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1981202

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则8：优先考虑查询语法（query syntax）而不是循环结构

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

C# 语言对不同控制结构都不乏支持： for， while， do/while 和 foreach，所有都是语言的一部分。从过去的计算机语言设计来看，很难不让人怀疑语言的设计者不是错过某些惊奇的循环结构。实际上，总是存在一个更好的方法：查询语法（query syntax）。

查询语法可以让你的程序逻辑从必要模式（imperative model）变为声明模式（declarative model）。查询语法定义了答案是什么而且决定了怎样得到这个答案的特殊实现。上面说到的整个点，这里指的是查询语法，方法调用语法带来的好处查询语句一样有。这点对于查询语句很重要，而且进一步扩展，方法语法实现查询表达模式，比 imperative 循环结构更加清晰表达你的意图。

这段代码用必要的方法填充数值然后打印内容到控制台：

```
int[] foo = new int[100];
for (int num = 0; num < foo.Length; num++)
    foo[num] = num * num;
foreach (int i in foo)
    Console.WriteLine(i.ToString());
```

即使上面的例子很小，但过于重视操作如何执行而不是什么操作执行。使用查询语法重新创建更可读的代码可以重复利用不同的编译块。

首先第一步，你使用查询结果改变数组的产生：

```
int[] foo = (from n in Enumerable.Range(0, 100)
select n * n).ToArray();
```

第二个循环你只要做类似的改动，尽管你需要写一个扩展方法对每个元素执行操作：

```
foo.ForAll((n) => Console.WriteLine(n.ToString()));
```

.NET BCL 使用 List<T> 实现 ForAll 。这只是简单地创建 IEnumerable<T> :

```
public static class Extensions
{
    public static void ForAll<T>(
    this IEnumerable<T> sequence,
    Action<T> action)
    {
        foreach (T item in sequence)
            action(item);
    }
}
```

这可能看起来没有意义，但是它更加能重复利用。任何你对数组元素的操作， ForAll 都能做到。

这个例子很简单，所以你不会看到很多好处。实际，你也许是对的。我们接着看其他不同问题。

很多问题要求你通过嵌套循环来完成。假设你需要从0到99产生所有(x,y)对。很明显你会使用嵌套循环：

```
private static IEnumerable<Tuple<int, int>> ProduceIndices()
{
    for (int x = 0; x < 100; x++)
        for (int y = 0; y < 100; y++)
            yield return Tuple.Create(x, y);
}
```

当然，使用查询你可以产生相同的对象：

```
private static IEnumerable<Tuple<int, int>> QueryIndices()
{
    return from x in Enumerable.Range(0, 100)
        from y in Enumerable.Range(0, 100)
        select Tuple.Create(x, y);
}
```

看起来很相似，但是查询更加简单即使这个问题的描述变得更困难起来。改变问题为产生的(x,y)对要求x和y的和小于100。比较这两个方法：

```
private static IEnumerable<Tuple<int, int>> ProduceIndices2()
{
    for (int x = 0; x < 100; x++)
        for (int y = 0; y < 100; y++)
            if (x + y < 100)
                yield return Tuple.Create(x, y);
}
private static IEnumerable<Tuple<int, int>> QueryIndices2()
{
    return from x in Enumerable.Range(0, 100)
           from y in Enumerable.Range(0, 100)
           where x + y < 100
            select Tuple.Create(x, y);
}
```

仍然很相近，但是 imperative 语法开始使用必须的语法产生结果而隐藏它的意义。所以再次改变问题。现在，条件一个条件：返回的数组必须按照它们到原点的距离排列。

```
private static IEnumerable<Tuple<int, int>> ProduceIndices3()
{
    var storage = new List<Tuple<int, int>>();
    for (int x = 0; x < 100; x++)
        for (int y = 0; y < 100; y++)
            if (x + y < 100)
                storage.Add(Tuple.Create(x, y));
    storage.Sort((point1, point2) =>
        (point2.Item1*point2.Item1 +
            point2.Item2 * point2.Item2).CompareTo( point1.Item1 * point1.Item1 + point1.
    return storage;
}
private static IEnumerable<Tuple<int, int>> QueryIndices3()
{
    return from x in Enumerable.Range(0, 100)
        from y in Enumerable.Range(0, 100)
        where x + y < 100
        orderby (x*x + y*y) descending
        select Tuple.Create(x, y);
}
```

有些细节很明显改变了。 imperative 版本更难去理解。如果你看得快，你几乎不会注意到比较函数的参数被对换了。那样保证排序的结果是降序的。没有注释或其他支持文档，imperative 代码很是很难读懂。

即使你发现 where 的参数被对换了，你会觉得这是错误么？ imperative 模式过于强调操作怎么样执行以至于很容易在这些操作中迷失什么操作正在完成的原意。

还有一个理由使用查询语法而不是循环结构：查询可以比循环结构创建更多组合的 API 。查询运费很自然构造算法块执行序列上的操作。查询的模式可以让开发者枚举序列中组合单一的操作为的组合操作。循环结构不能有类似的组合。你必须临时存储每步的结果，或创建对序列上组合操作的方法。

上个例子就体现了这点。操作组合自 一个过滤操作（where块），一个排序操作（sort块）和一个选择操作select块）。这些操作全部在一个枚举操作完成。 imperative 版本创建历史存储模型而且排序操作被分离出来。

我几经讨论过查询语法，但是你应该记住每个查询操作都有相应的方法调用语法。有时候查询语法更自然，有时候方法调用语法语法更自然。在上面例子，查询语法更加可读。下面是对应的方法调用语法：

```
private static IEnumerable<Tuple<int, int>> MethodIndices3()
{
    return Enumerable.Range(0, 100).
        SelectMany(x => Enumerable.Range(0,100),
        (x,y) => Tuple.Create(x,y)).
        Where(pt => pt.Item1 + pt.Item2 < 100).
        OrderByDescending(pt =>
            pt.Item1* pt.Item1 + pt.Item2 * pt.Item2);
}
```

查询语法或方法调用语法哪个更加可读是一个风格问题。在这个例子，我相信查询语法是更清晰的。然而，其他例子可能会不同。此外，一些方法是没有对应的查询语法的。向 Take，TakeWile， Skip， SkipWhile， Min， 和 Max 在一定程度上需要你使用方法语法。其他语言，特别是 VB.NET 定义了更多查询语法的关键字。

有的人老是会认为我们讨论的这部分即查询的性能会比循环更慢。虽然你可以用一个简单例子说明循环能跑赢查询，但这不是一般地规则。如果你有一个特别的例子即查询结果表现的不够好，你才需要决定测量性能。然而，在你完成重写一个算法之前，考虑使用 LINQ 的并行扩展。使用查询语法的另一个好处是使用 .AsParallel() 方法可以并行执行查询。（查看原则 35）。

C#开始是一个 imperative 语言。它几下保留了属于这个遗产的所有特性。自然你可随意使用最熟悉的工具（循环结构）。然而，这些工具可能不是最好的工具。当你发现你自己写着循环结构的形式，问下自己是否使用查询来写。如果查询语法不行，考虑使用函数调用语法。在大多数情况，你会发现创建比使用 imperative 循环结构更清晰的代码。

小结：

这篇文章讲的点还是比较好的，至少是一种新的编程模式（对于像D.S.Qiu这样的人来说），可以尝试多使用些，会带来一些便利。本来一直很纠结 imperative model 和 declrarive model 的中文翻译，由于对计算机语言的历史都不了解，对一些专业名称很没有太多认识，甚至都没有怎么接触，只有困惑来了就去看下 wikipedia 。每天都是到这个时候才写完，脖子今天一直很不舒服，不要有颈椎病呀，完了。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1982137

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则9：在你的 API 中避免转换操作

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

转换操作引入不同类之间的替代性。替代性的意思是一个类可以被另外一个类代替。这有一个好处：一个子类可以代替基类，像下面 shape 继承结构给出例子。你定义基类 Shape 并自定义它的子类： Rectangle， Ellipse， Circle 等。在期望是 Shape 的地方你可以用 Circle 代替它。这是使用多态的替代性。这个是因为 Circle 是 Shape 的一个特别类型。当你创建一个类，某些转换是自动被允许的。任何对象都可以替代 .NET 类结构的基类 System.Object 的对象。相同的原理，你创建的任何类的对象都可以隐式代替它实现的接口，或者任何基类的接口，或者任何基类。语言还支持一系列的数值转换。

当你定义类的转换操作符，等于告诉编译器你的类型可以代替目标类型。这些转换经常会导致一些微妙错误因为你的类型可能不是能很完美替代目标类型。有一个副作用就是当你修改目标类型的状态可能不会对你的类型产生相同的影响。更糟糕的是，如果你的转换操作符返回一个临时对象，这个临时对象将会被垃圾回收而永久丢失。应用转换操作符的规则是基于编译时的类型，而不是运行时的对象的类型。使用你的类可能需要执行多次转换操作符，这个实践会导致代码很难维护。

如果你想转换任意类型到你的类型，使用构造器。这个更清晰的反应了创建对象的行为。转换操作符会在代码中引入很难发现的问题。假设你的代码的库的继承结构如图1.1一样。Circle 和 Ellipse 是 Shape 的子类。你打算不考虑这个结构关系，因为你认为，即使 Cirle 和 Ellipse 是相关的，但你不想要结构中非抽象的兄弟关系，当你尝试从 Ellips 类对象得到 Circle 对象就会发现几个实现问题。然而，你实现的每个 Circle 对象都可以是一个 Ellipse 对象。此外，一些 Ellipse 对象可以代替 Circle 对象。

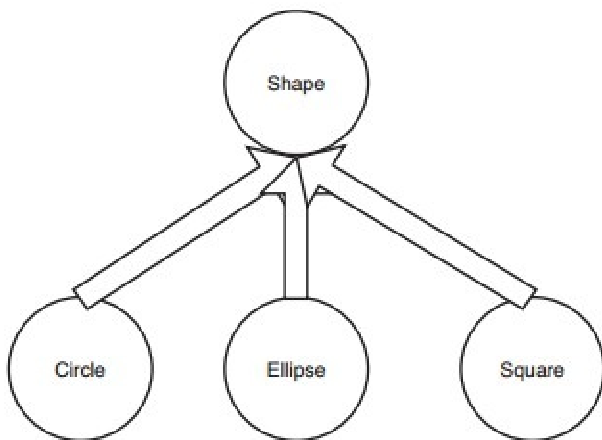


Figure 1.1 Basic shape hierarchy.

这导致你添加两个转换操作符。Circle 对象都是 Ellipse 对象，所以你需要添加一个隐式转换从 Circle 创建一个 Ellipse 对象。当一个类型需要转换到另一个类型隐式转换都会被调用。相反，显式转换只有在程序员在代码中强制转换才会被调用。

```
public class Circle : Shape
{
    private PointF center;
    private float radius;
    public Circle() : this(PointF.Empty, 0)
    {
    }
    public Circle(PointF c, float r)
    {
        center = c;
        radius = r;
    }
    public override void Draw()
    {
        //...
    }
    static public implicit operator Ellipse(Circle c)
    {
        return new Ellipse(c.center, c.center,c.radius, c.radius);
    }
}
```

既然你已经有一个隐式转换操作符，你可以任何期望是 Ellipse 的地方使用 Circle 。此外，这个转换是自动发生的：

```
public static double ComputeArea(Ellipse e)
{
    // return the area of the ellipse.
    return e.R1 * e.R2 * Math.PI;
}
// call it:
Circle c1 = new Circle(new PointF(3.0f, 0), 5.0f);
ComputeArea(c1);
```

这个例子就是我说的代替：一个 Circle 对象可以代替 Ellipse 对象。ComputeArea 函数甚至能在代替后工作。你获得好运气。但是研究下这个函数：

```
public static void Flatten(Ellipse e)
{
e.R1 /= 2;
e.R2 *= 2;
}
// call it using a circle:
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten(c);
```

这就不能工作了。 Flatten 方法需要 Ellipse 的参数。编译器会某些情况下会将 Ellipse 转换为 Circle 。你定义的隐式转换就是实现这个工作的。你的转换会被调用，而且 Flatten 函数接受的被隐式转换创建的 Ellipse 对象。这个临时对象呗 Flatten 函数修改，并且立即变成了垃圾。副作用预期是从你的 Flatten 函数发生的，但是仅仅是一个临时变量。最后的结果就是 Circle c 没有发生任何事情。

将隐式转换该为强制转换：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten((Ellipse)c);
```

原来的问题还是存在。你只是强制你的使用添加强制转换来引起这个问题。你还是创建临时对象， Flatten 函数作用在这个临时对象上，并且丢失这个对象。 Circle 根本没有被改变。相反，如果你创建一个构造器转换 Circle 到 Ellipse ，这个操作就更清晰了：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten(new Ellipse(c));
```

大多数程序员会看到上面两行代码而立即发现传给 Flatten() 的 Ellipse 的任何修改都会丢失。他们会持有一个新的对象来修复这个问题：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten(c);
// Work with the circle.
// ...
// Convert to an ellipse.
Ellipse e = new Ellipse(c);
Flatten(e);
```

变量持有了被 Flatten 修改的 Ellipse 对象。通过构造函数代替转换操作符，你还没有失去任何功能，你只是创建新对象就使它更清晰。（老练的 C++ 程序员，应注意， C# 调用构造函数不会进行隐式或显式转换。您可以创建只有当你明确地使用 new 运算符的新对象，并在没有其他时候。所以在 C# 构造器中不需要 explicit 关键字。）

转换操作符你使得对象返回原来没有的行为的域。这会其他一些问题。你隐藏了一个大漏洞在封装的类中。当强制你的类型转换到另一对象时，使用你这个类的可以访问内部变量。在原则26中讨论了最好避免的所有原因。

转换操作符引入了代替的一个方式但会以前你的代码抛出问题。你必须明确：用户预期的类可以在你创建的这个类的对象任何地方使用。当对象的代替可用，你使得调用者用的是你创建的临时对象或能访问内部域。这个微妙的错误很难被发现因为编译器产生了转换对象的代码。在你的 API 里避免转换操作。

小结：

这个原则虽然很简短但却很精辟，告诉我们实际编程应该更多通过规范来约束代码的行为，而不是将诸多情况都交给编译器来处理，这样总会被自己坑到的，要能在代码中严格控制自己的逻辑。周末还有好多工作，加油！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1983118

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则10：使用默认参数减少函数的重载

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

C# 现在已经在函数调用上支持命名参数了。这意味着名字形式的参数是你的类 public 接口的一部分。改变 public 参数的名字将破坏函数调用代码。这意味着很多时候要避免使用命名参数，或者避免改变 public 或 protected 方法的命名参数的名字。

当然，没有一个语言设计者增加的特性会增加你的难度。命名参数和默认参数防止了很多 API 的多而杂乱，特别是 Microsoft Office 的 COM API 。下面这段代码使用 COM 类的方法创建 Word 文档并插入一小段文本：

```
var wasted = Type.Missing;
var wordApp = new Microsoft.Office.Interop.Word. Application();
wordApp.Visible = true;
Documents docs = wordApp.Documents;
Document doc = docs.Add(ref wasted, ref wasted, ref wasted, ref wasted);
Range range = doc.Range(0, 0);
range.InsertAfter("Testing, testing, testing. . .");
```

这段代码使用了小而没用的 Type.Missing 对象次数。很多 Office Interop 应用都会使用大量的 Type.Missing 对象。这些实例弄混了你的应用并隐藏了你构建软件的实际逻辑。

在 C# 语言中，这个混乱只要是因为添加了默认参数和命名参数导致的。默认参数意味着意味着这些 Office API 可以使用默认值对那些所有使用 Type.Missing 的地方。简化上面的小段代码：

```
var wordApp = new Microsoft.Office.Interop.Word.Application();
wordApp.Visible = true;
Documents docs = wordApp.Documents;
Document doc = docs.Add();
Range range = doc.Range(0, 0);
range.InsertAfter("Testing, testing, testing. . .");
```

命名参数的作用是在任何有默认参数的 API 中，你只需要制定你想要使用的参数。这个简单并省去了多次重载。实际上，四个不同参数，你需要重载15个不同的 Add() 来达到命名参数和默认参数相同的灵活性。记住一些 Office API 有多达16个参数，默认参数和命名参数就是一个很大帮助。

我在参数列表使用 ref 修饰符，但在 C# 4.0的另一个改变就是在 COM 中这是默认的。这是因为 COM ，一般来说，使用对象引用来传递参数，所有几乎所有参数都是按引用来传递参数的，即使函数内没有对参数进行改变。实际上， Range() 调用通过引用传入值(0,0)的。我们没有包含 ref 修饰符，因为那将是明显的误导。实际，在多少代码中，我们都没有包含 ref 修饰符去调用 Add() 。上面我就是这样做的，你可以去查看下实际的 API 签名。

当然，仅仅是因为 COM 和 Office API 的命名参数和默认参数，这不意味着你需要限制使用它们在 Office Interop 应用中。实际上，这是不能的。开发者使用命名参数调用你的 API 无论你是否希望他们这么做。

下面的方法：

```
private void SetName(string lastName, string firstName)
{
    //elided
}
```

使用命名参数调用避免命名参数次序的混乱：

```
SetName(lastName: "Wagner", firstName: "Bill");
```

注解命名参数保证别人后面看这段代码不用去想参数的位置是否正确。开发者使用命名参数将阅读代码的人的更清晰。当调用的函数有多个相同类型的参数，使用命名参数会使你的代码更加具有可读性。

改变参数名字的破坏会表现得很有趣。在 MSIL 中参数名字存储在函数调用点，而不是函数入口。你可以改变参数名字并且发布的组件不会破坏任何使用这个组件的代码。使用这个组件的开发者会有一个重要的改变当需要重新编译升级版本，但之前的版本的客户端代码仍然可以正确运行。所以你至少不会破坏已存在的程序集。使用你代码的开发者会懊恼，但是不会对你抱怨出现的问题。例如，假设你修改 SetName() 的参数名字：

```
public void SetName(string Last, string First);
```

你将会编译和发布这个程序集作为一个补丁。任何调用这个方法的程序集都能继续运行，即使包含了指定参数名字的 SetName 的调用。然而，当客户端开发者想要升级他们的程序集，像下面的代码都不会通过编译：

```
SetName(lastName : "Wanger",firstName : "Bill");
```

改变默认参数的值童谣需要调用者重写编译保证这个更新一致。如果你编译你的程序集并发布这个补丁，所有已存在的调用将继续使用之前的默认值。

当然，你也不会随意的对使用你的组件的开发者带来麻烦。所以，你必须把参数的名字当做组件 public 接口的一部分。改变参数名字在客户端代码再编译时会有问题。

此外，增加参数（即使是默认参数）都会在运行时有问题。默认参数的实现跟命名参数是类似的。调用点会包含 MSIL 用于反射出默认值的存在和具体值时多少的注解。函数入口替换调用者没有明确指定的默认参数的值。

所以，即使增加一个默认参数，都会在运行时出现问题。如果它们有默认参数，在编译时不会有问题。

至此，经过一番解释，这个指导应该更加清晰了。为了初始版本的发布，使用默认参数和命名参数可以创造使用者想要得到重载组合。但是，一旦你创建未来的版本，你必须创建额外参数的重载。那样已经存在的客户端程序会继续有效。此外，在任何将来的版本中，避免更改参数名称。它们现在是你的 public 接口的一部分。

小结：

这篇跨了一个星期，因为自己脖子到现在还是疼的，然后公司搬家，这个星期脖子一直都很痛，加上上班的工作任务也重，会到家就直接洗澡休息了，连续休息了一周感觉还是不错的，至少脸上的“平静”了很多，气色也好了很多，但是又觉得很不心安，感觉荒废了，我要加油，继续努力。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1986873

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则11：理解小函数的魅力

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

作为一个有经验的程序员，不管在喜欢 C# 之前用的是什么语言，都会积累开发更高效代码的经验。有时，能在之前的环境有效的方法在 .NET 环境中会起反作用。当你想手动优化 C# 编译器的算法时就会体会到。你的行为经常会阻止 JIT 编译器更高效的优化。你额外提升性能的工作实际会产生更慢的代码。你最好的让你的代码写的尽可能地清晰。让 JIT 做剩下的工作。最常见的一个例子就是创建一个更长更复杂的函数希望能避免函数调用的过早优化会引起问题。像这样组装函数逻辑到一个循环体中实际伤害了 .NET 程序的性能。这是很直观的，所以下面我们对细节进行详细说明。

.NET 运行时调用 JIT 编译器翻译 C# 翻译器产生的的 IL 代码为机器码。这个过程是评价摊销在程序执行的整个生命周期。在程序启动时， JITing 不会全部载入你的程序，而是 CLR 是基于一个一个函数调用 JITer 。这最大限度地减少启动时的开销至一个合理的水平上。但如果更多代码需要被 JIT 程序会变得更迟钝。函数只有被 JIT 后才能被调用。你可以通过拆分代码成更多更小的函数而不是直接使用更少而大的函数来最大减小被 JIT 产生额外的代码数量。考虑下面的例子，尽管有点刻意：

```
 public string BuildMsg(bool takeFirstPath)
{
    StringBuilder msg = new StringBuilder();
    if (takeFirstPath)
    {
        msg.Append("A problem occurred.");
        msg.Append("\nThis is a problem.");
    msg.Append("imagine much more text");
    }
    else
    {
        msg.Append("This path is not so bad.");
        msg.Append("\nIt is only a minor inconvenience.");
        msg.Append("Add more detailed diagnostics here.");
    }
    return msg.ToString();
}
```

BuildMsg 第一个调用，两个分支代码都会被 JIT 。但只有一个是需要的。但是假设你按下面方式重写函数：

```
public string BuildMsg2(bool takeFirstPath)
{
    if (takeFirstPath)
    {
        return FirstPath();
    }
    else
    {
        return SecondPath();
    }
}
```

不过，这个例子有点牵强，而且不会有太大的区别。但是考虑下你经常写的更广泛的例子：一个 if 的结构有超过 20 的条件状态。函数第一个载入是你需要为所有分支花费 JIT 的开销。如果你分支不大可能是错误条件，你可以很容易避免这个开销。短小的函数意味着 JIT 编译器只编译逻辑上需要的函数，而不是你没有立即使用整个一长串代码。 switch 结构可以节省好几被的 JIT 开销，如果每个 case 条件定义为 inline 而不是 单独的函数。

短小而简单的函数可以让 JIT 编译器很容易支持寄存器化（enregistration） 。寄存器化是指处理器选择寄存器而不是栈存储局部变量。创建更少的局部变量使得 JIT 编译器更好的找到可用的寄存器。控制流的简化同一会影响 JIT 编译使用寄存器存储变量。如果函数有一个循环体，这个循环变量很可能存储在寄存器中。然而，当你在一个函数中创建了好几个循环体，JIT 编译就要做一个艰难地选择哪一个循环变量存在寄存器中。越简单越好。一个简短的函数很可能包含更少的局部变量，可以使得 JIT 编译器更容易使用寄存器优化程序。考虑下面的例子：

```
// readonly name property:
public string Name { get;private set;}

// access:
string val = Obj.Name;
```

属性的访问器比函数调用包含更少的指令：保持寄存器状态，执行开始很结尾的代码，存储函数的返回值。如果需要有参数的话，还需要在栈上压人参数这一个步骤。如果使用 public 域的用到指令会变得更少。

当然，你不会那样做，因为你知道尽少使用 public 数据成员（原则1）。 JIT 编译器知道你需要兼顾代码的效率和优雅，所以属性的访问器是被内联调用的。当速度和大小的好处有利于用函数体代替函数调用时， JIT 才会将函数内联调用。该标准没有定义任何额外的内联规则，在将来任何实现都可能改变。而且，内联函数不是你的责任。 C# 语言甚至没有提供关键用以提示编译器某个方法应该被声明为内联。 C# 编译器不会通过任何 JIT 有关内联的提示。（你可以使用 System.Runtime.CompilerServices.MethodImpl 特性，指定方法不被内联。这是在调试时在函数调用栈保留函数名字的典型做法。

```
[MethodImpl(MethodImplOptions.NoInlining)]
```

所有你需要做的就是保证你的代码尽可能的清晰，使得 JIT 编译器可以很容易做最好的决定。现在这个建议应该变得更熟悉：小函数优于更容易被内联调用。但是记住虚函数或包含 try/catch 块的函数式不会被内联的。

内联修改了执行代码被 JIT 的原则。再看下 Name 属性的访问：

```
string val = "Default Name";
if (Obj != null)
    val = Obj.Name;
```

如果 JIT 编译器将属性访问器置为内联，这必须会 JIT 包含属性调用的方法的代码。

建议构建更小，组合的方法在LINQ查询和函数式编程的世界更为重要。所有的LINQ查询方法是相当小的。此外，大多数传递给LINQ查询的谓词，行为，和函数都是很小的代码块。小的，更可组合自然意味着这些方法，行为，谓词和函数更容易重用。此外， JIT 编译器更有机会优化代码使得在运行时执行的更有效率。

你不要负责决定最好机器级展示你的算法。 C# 编译器和 JIT编译器以前为你做这个。 C# 为每个方法产生 IL 代码， JIT 在宿主机器上将 IL 代码方法为机器码。你不用关心 JIT 编译器在不同情况下使用的准确规则，这些将随着时间的推移开发出更好的算法。相反，你应该关心你的表达

算法的方式，使得它容易在环境中的工具做最好的工作。幸运的是，这些规则您已经遵循良好的软件开发实践的规则是一致的。更多的时候：更小，更简单的函数。

JIT 编译器同时会对是否使用内联函数做决定。内联意味着用函数体代替函数的调用。

因为不同分支代码被分解成各自的函数，这些函数只有在需要的是才被 JIT 而不是第一次 BuildMsg 被调用时。

记住翻译 C# 代码到机器可执行码需要两个步骤： C# 编译器产生发布 在程序的 IL 代码。 JIT 编译器根据需要为每个方法（或者是内联调用的函数组）产生机器码。小函数使得 JIT 编译器可以更容易分摊开销。小函数还更有可能使用内联方式调用。这不只是短小：简单的控制流会出现更多问题。较少的控制分支可以让 JIT 编译器使用寄存器变量。这不仅是一个让你代码写的更加清晰的好方法，这更是如何创建再运行时更加高效的代码

小结：

终于写翻译完第一章共11个原则，哎，晚上脖子两侧开始疼了，希望会没事的。经常容易心情不好，前几天一个朋友的鼓励：就算失去所有，不是还有明天吗，虽然我现在已经不需要心灵鸡汤了，但其中的鼓励还是可以感知到的，也不早了（又一点了）

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1986910

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

第二章 .NET 资源管理

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

一个很简单的事实，.NET 程序运行在托管的环境有对创建有效 C# 的设计有很大影响。想要利用这个环境的优势需要你的思维从其他环境的转换到 .NET 公共语言运行库（CLR）上去。这意味着你需要理解 .NET 垃圾回收器。.NET 内存管理环境的概述对于理解本章的具体建议还是很有必要的，所以我们从这个概述开始。

垃圾回收器（GC）为你控制托管内存。不像自然环境，你不需要负责大多数的内存泄露，悬浮指针，未初始化指针，或者其他一些了内存管理问题。但是垃圾回收器没那么神奇：你也需要自己清理。你要负责非托管资源，例如文件句柄，数据库连接，GDI+对象，COM对象，和其他系统对象。此外你可能引起对象停留在内存比你想象的久因为你使用 event 处理或 delegate 创建它们之间的连接。查询，它执行等待结果，也会引起对象保持引用时间比你期望的更长。在闭包中捕获约束变量，并且这些约束变量会可以一直可访问直到已经离开了这个结果的作用域。

好消息是：因为 GC 控制内存，明确的设计会更容易实现。无论是循环引用，还是简单关系和复杂的 web 对象，都是更容易的。GC 的

标记和压缩算法可以高效检测到这些关系和完全移除不可达 web 对象。 GC 决定对象是否可达是通过对象树形结构从根部开始漫游，而不是强制跟踪每个对象的引用，COM 就是这样的。 EntitySet 类提供了例子，它的算法简化了对对象关系的判定。实体是从数据数据库加载的对象集合。每个实体可能包含其他实体对象的引用。这些实体还可以能包含对其他实体的链接。就像关系数据库的实体集模型，这些链接和引用都是可循环的。

有些引用是展示不同 EntitySet 的 web 对象。释放内存是 GC 的责任。因为 .NET 框架设计者不需要释放这些对象，web 对象引用的释放的复杂性不会构成问题。不需要决定 web 对象的合理释放次序，这是 GC 的工作。GC 的设计简化了识别这类 web 对象为垃圾的问题。应用可以停止引用任何实体就是垃圾。垃圾回收器会知道是否这个实体仍然由应用中活着的对象可到达。应用中任何不可达到的对象都是垃圾。

垃圾回收器运行自己的线程去移除程序中没有用的内存。它每次也会压缩托管的堆内存。压缩堆是通过移动活着的对象到一个托管的堆中以使得未使用的内存在一个连续的内存块。图 2.1 就是垃圾回收前和后堆内存的截图对比。GC 处理完所有空闲内存都被放在连续的块中。

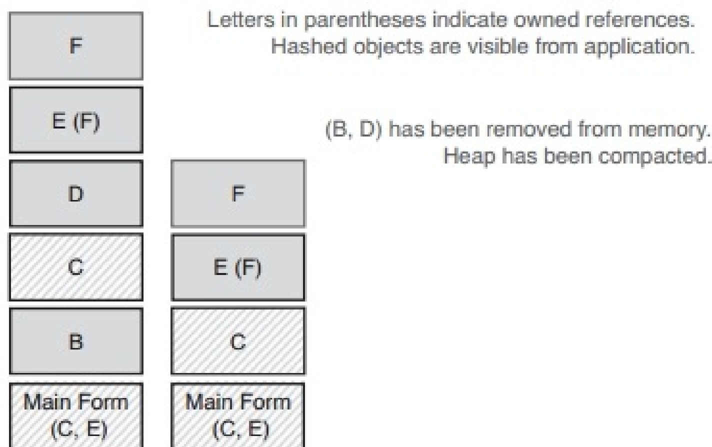


Figure 2.1 The Garbage Collector not only removes unused memory, but it also moves other objects in memory to compact used memory and maximize free space.

就像你刚学到的，内存管理（堆内存的管理）完全是垃圾回收器的责任。其他系统的资源必须有开发者管理：你和使用你的类的人。两种机制帮助开发者控制的非托管资源的寿命：析构函数和 IDisposable 接口。析构函数是一个被动的机制确保你的对象总是有方式释放非托管资源。析构函数会有很多缺点，所有你也可以实现 IDisposable 接口提供几乎不入侵的方式及时返回资源给系统。

析构函数被垃圾回收器调用。它们会在对象变成垃圾之后某个时间被调用。你不知道什么时候调用。你只能知道的是如果某个时间被调用了你的对象就不可到达。这是跟 C++ 很大的不同，并且对你的设计有很重要的影响。要经验的 C++ 程序员编写类总是在构造函数分配重要资源然后在析构函数释放：

```
// Good C++, bad C#:
class CriticalSection
{
    // Constructor acquires the system resource.
    public CriticalSection()
    {
        EnterCriticalSection();
    }
    // Destructor releases system resource.
    ~CriticalSection()
    {
        ExitCriticalSection();
    }
    private void ExitCriticalSection()
    {
        throw new NotImplementedException();
    }
    private void EnterCriticalSection()
    {
        throw new NotImplementedException();
    }
}

// usage:
void Func()
{
    // The lifetime of s controls access to
    // the system resource.
    CriticalSection s = new CriticalSection();
    // Do work.
    //...
    // compiler generates call to destructor.
    // code exits critical section.
}
```

通常的 C++ 习惯是保证资源回收是没有异常。这在 C# 是行不通的，至少，不是同样的方式。确定的析构函数不是 .NET 环境或 C# 语言的一部分。在 C# 语言尝试强制 C++ 习惯的确定的析构函数不会很好的奏效。在 C# ，析构函数最后才执行，但它不会及时执行。在上面的例子中，代码最后会退出临界区，但是在 C# 中，函数退出后它不会退出临界区。那会在后面确定的时间发生。你不可能知道什么时候。析构函数只是保证对象申请的非托管资源会最终释放。但析构函数执行没有确定的时间，所以你的设计和编码实践应该尽量减少析构函数的创建，同时也尽量减少析构函数的执行如果它存在的话。在本章中你将学习到什么时候你一定要创建析构函数，以及如何减少有析构函数的负面影响。

依赖于析构函数还会引入性能的损失。需要析构函数对象会被垃圾回收器消耗一部分性能。当 GC 发现对象是垃圾但是需要执行析构，它不能将对象理解从内存中移除。首先，它调用析构函数。析构函数不是在和垃圾回收器同一个线程中执行的。而是， GC 把每个等待析构的对象放进一个队列中并且起另一个线程去执行所有析构函数。它会持续进行，把垃圾移除。在下一次 GC 循环时，已经被析构的对象就会被移除内存。图2.2 显示3种 GC 操作和不同内存使用。注意到需要析构的对象会留着内存多些循环周期。

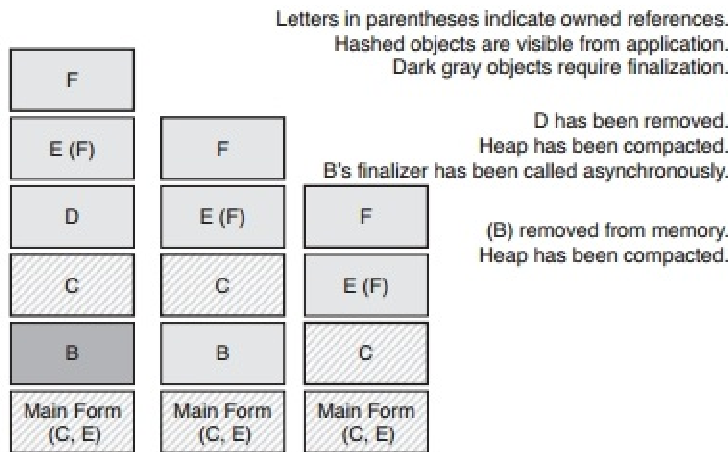


Figure 2.2 This sequence shows the effect of finalizers on the Garbage Collector. Objects stay in memory longer, and an extra thread needs to be spawned to run the Garbage Collector.

这可能会让你认为需要析构的对象会在内存比一般对象多待一个 GC 周期。我只是简化地描述。它是比我描述的更复杂因为 GC 的设计决策。.NET 垃圾回收器定义代来优化工作。代帮助 GC 更快确定最有可能是垃圾的候选对象。从上次垃圾回收操作创建的对象都是0代对象。在一次 GC 操作后存活下来的就是1代对象。经过2次或更多 GC 操作存活的是2代对象。代的目的是区分局部变量和生命周期是整个应用的对象。0代的对象大多数都是局部对象。成员变量和全家变量很快会进入1代而且最后进入2代。

GC 通过现在检查1代和2代对象的频率来优化工作。每次 GC 循环都会检查0代对象。粗略假设 GC 会10次检查0代和1代对象。而要超过100次检查所有对象。考虑析构以及它的开销：需要析构的对象要比不需要析构的对象多待超过9个回收循环。如果仍然没有被析构，它将进入2代。在2代，对象会生存上100个循环知道下次2代回收。我已经花了一些时间解释为什么析构函数不是一个好的解决方案。但是，你仍需要释放资源。解决这些问题你可以使用 IDisposable 接口和标准回收模式（查看本章原则17）。

在最后，记住托管环境垃圾回收器会负责内存管理，最大的好处是：内存泄露和其他指针相关的问题不再是你的问题。非内存资源你要强制创建析构函数来保证正确清理那些非内存资源。析构函数会比较大影响你程序的性能，但是你必须实现它以避免内存泄露。实现和使用 IDisposable 接口避免析构函数引入的垃圾回收的性能消耗。下一节将进入具体的原则，帮助您创建程序，更有效地利用环境。

小结：

终于翻译完第二章了，不断不断坚持，还是觉得要完成，本来凌晨2:00就要结束第二章的战斗的，后面状态还是没有调整过来，从昨天翻译有20页，虽然现在感觉印象还不是特别深刻，但是至少有用还是可以明确感受到。

附上第二章的目录：

- 原则14：减少重复的初始化逻辑
- 原则15：使用 using 和 try/finally 清理资源
- 原则16：避免创建不需要的对象
- 原则17：实现标准的 Dispose 模式
- 原则18：值类型和引用类型的区别
- 原则19：确保0是值类型的一个有效状态
- 原则20：更倾向于使用不可变原子值类型

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2079806

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则12：选择变量初始化语法（initializer）而不是赋值语句

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明**出处**：http://dsqiu.iteye.com

类经常会有多个构造函数。时间一长，成员变量和构造器很难保持同步。避免这个的最好方式就是在变量声明的时候初始化而不是在构造函数内初始化。你应该使用初始化语法为静态变量和实例变量进行初始化。

在 C# 中，当你声明变量的时候构建变量是很自然的。当你声明变量时直接进行初始化：

```
public class MyClass
{
        // declare the collection, and initialize it.
        private List<string> labels = new List<string>();
}
```

无论 MyClass 类有多少个构造函数， labels 都会被正确初始化。编译器会在构造器的开头为你定义的实例成员变量生成初始化代码。当你增加一个构造函数， labels 就会获得初始化。同样的，如果你增加一个新的成员变量，你不需要在每个构造函数中添加初始化代码，在定义的地方初始化变量是非常合适的。同样重要的是，初始化语句（initializer①）也会被添加进编译器产生的默认构造函数中。当你的类没有显式定义任何构造函数编译器会创建一个默认构造函数。

初始化语句比构造函数中的语句更方便快捷。初始化语句被放在你的构造函数之前。初始化语句在基类构造函数之前执行，而且它们执行的顺序是定义它们的顺序。

使用初始化语句是你的类中成员变量未初始化最简单的方法，但不完美。有三种情况，你不应该使用初始化语句语法。第一种情况是当你要将对象初始化为0或 null ，系统在你的代码执行之前，会默认为所有内容设置为0。系统产生的0初始化是使用非常低级的 CPU 指令设置整块内存为0。你的任何额外的0初始化都是多余的。 C# 编译器会非常尽责地添加额外指令去又一次设置内存为0。这没有问题——只是低效的。实际上，如果是值类型，那会更低效。

```
MyValType myVal1; // initialized to 0
MyValType myVal2 = new MyValType(); // also 0
```

上面两条语句的变量都会初始化为0。第一个是设置 myVal1 的内存为0。第二个是使用 IL 指令 initObj ，这个会引起 myVal2 的封箱和拆箱操作。这需要相当的额外时间（查看原则45）。

当你对同一个对象进行多次初始化时，第二种情况就来了。只有变量在所有构造函数中接收的初始化代码是一样的才使用初始化语句语法。下面版本的 MyClass 构造函数呢中创建两个不同的 List 对象：

```
public class MyClass2
{
    // declare the collection, and initialize it.
    private List<string> labels = new List<string>();
    MyClass2()
    {
    }
    MyClass2(int size)
    {
    labels = new List<string>(size);
    }
}
```

当你创建 MyClass2 对象，并指定了集合的大小，你就创建了两个数组队列。其中一个立即变为垃圾。变量初始化语句在每个构造函数之前执行。构造函数中创建了第二个数组队列。编译器产生了这样的 MyClass2 版本，当然你是绝不会手动这样写的。（查看原则14，使用恰当方式处理这个问题）。

```
public class MyClass2
{
        // declare the collection, and initialize it.
        private List<string> labels;
        MyClass2()
       {
               labels = new List<string>();
       }
       MyClass2(int size)
      {
              labels = new List<string>();
              labels = new List<string>(size);
      }
}
```

如果你使用隐式属性也是会出现同样的情况（查看原则1）。你的代码不能访问编译生成的备份存储域，所以隐式属性就不能使用初始化语句。除了在构造函数汇总初始化隐式属性，你别无选择。使用隐式属性仍然具有的一个好处就是在 set 访问器中不用进行逻辑校验。如果你把隐式属性迁移为命名的备份存储域，你应该使用初始化语法而不是构造函数来更新初始化代码。因为使用隐式属性是数据成员的更好选择，原则14说明了如何减少隐式属性数据域的复制。

要在构造函数中初始化的最后一个原因是方便异常的处理。你不能再初始化语句中使用 try 块。初始化成员变量时的任何异常都会传给实例化的对象。你不能在类内对这个异常进行捕获。你需要把初始化代码移到构造函数中，这样就可以实现正确的回收代码来创建你的类型和妥善处理异常（查看原则47）。

成员变量初始化语法是保证是初始化的无论哪个构造函数被调用的最简单方法。初始化语法都在每个构造函数之前执行。使用这个语法意味着你不会因新版本中添加的构造函数而忘记初始化。当所有构造函数以相同方式创建成员变量时，使用初始化语法；这易于阅读，更易

于维护。

①：initializer 翻译为初始化语法

小结：

昨晚一直状态不好，就没有写下去，还好这个原则比较简短，就借用早餐+午休的时间完成了。这个原则建议我们要使用初始化语法来初始化成员变量，这样既简单又便于维护，不过除了上面说的三种情况之外。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/1987663

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则13：使用恰当的方式对静态成员进行初始化

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

你应该知道静态成员变量在创建对象实例之前就已经初始化了。C# 提供了静态初始化语法和静态构造函数对静态成员变量进行初始化。静态构造函数是一个比其他函数，变量，属性在没有访问之前就被执行的特殊函数。你可以使用这个函数初始化静态变量，完善单例模式，或者任何需要在类还没有被使用的必要工作。你不能使用构造函数，一些特殊 private 函数，或其他语法初始化静态变量。

和实例初始化一样，初始化语法是静态构造函数的另一选择。你只需要声明一个静态成员，然后使用初始化语法。如果有静态变量的初始化逻辑很复杂，你可以创建一个静态构造函数。

C# 实现单例模式最常用的方法就是静态构造函数。让构造函数变为 private，然后添加一个初始化语法：

```
public class MySingleton
{
    private static readonly MySingleton theOneAndOnly = new MySingleton();
    public static MySingleton TheOnly
    {
        get { return theOneAndOnly; }
    }
    private MySingleton()
    {
    }
    // remainder elided
}
```

单例模式很容易像上面的方式一样实现。如果初始化逻辑更复杂：

```
public class MySingleton2
{
    private static readonly MySingleton2 theOneAndOnly;
    static MySingleton2()
    {
        theOneAndOnly = new MySingleton2();
    }
    public static MySingleton2 TheOnly
    {
        get { return theOneAndOnly; }
    }
    private MySingleton2()
    {
    }
    // remainder elided
}
```

和实例初始化一样，静态初始化语法在任何静态构造函数之前执行。并且，你的静态初始化语法比基类的静态构造函数更早执行。

CLR 在类型第一次在应用空间被访问之前自动执行静态构造函数。你只能定义一个静态构造函数，并且不能有参数。因此你需要注意在静态构造函数是否会产生异常。如果你在静态构造函数抛出了异常， CLR 会终止你的程序。这种情况捕获异常是非常阴险的。这时的代码区创建这个类型就会失败直到 AppDomain 被卸载。 CLR 就不能执行静态构造函数去初始化这个类型。即使你再尝试，类型不会被正确初始化。创建这个类型的对象（或者任意子类）都是没有被定义的。所以，这是不允许的。

异常是使用静态构造函数代替静态初始化语法的最常见原因。因为你使用静态初始化语法，你不能自己捕获异常。使用静态构造函数，你就可以（查看原则47：）：

```
 static MySingleton2()
{
    try
    {
        theOneAndOnly = new MySingleton2();
    }
    catch
    {
        // Attempt recovery here.
    }
}
```

静态初始化语法和静态构造函数是最干净，最清晰的方式去初始化静态成员变量。它们具有很强的可读性和很容易正确使用。它们被支持就是为了解决这个其他语言初始化静态成员会出现的问题。

小结：

这个事情一直拖着，一直想继续，今天又起了个头了，希望能一鼓作气。翻译很烂并且有很多不明确的地方（自己读的疑惑），算是给自己买下一个伏笔吧，还有很多工作要去完善。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2077189

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则14：减少重复的初始化逻辑

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

写构造函数是一个反复的工作。很多开发者总是写了第一个构造器然后复制粘贴代码到另外一个构造器，以满足多个重载函数接口的定义。但愿，你不是其中的一个。如果你是，那么请停止这么做。老练的 C++ 程序员会提取出通用的算法成为一个 private 辅助方法。但是，还是请停止那样做。如果你发现多个构造函数包含相同的逻辑，将这个逻辑提取到一个通用的构造器中。这样做的好处是，你可以避免代码复制，构造器初始化会产生更高效的代码。C# 编译器将构造函数识别为特殊的语法，进而移除重复的变量初始化和重复基类构造函数的调用。这样使得最后对象执行最少的代码去进行初始化。同时你因把这个任务交个通用构造函数写的代码是最少的。

构造初始化允许一个构造函数调用另一个构造函数，下面的这个用法的例子：

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Name of the instance:
    private string name;
    public MyClass() :
        this(0, "")
    {
    }
    public MyClass(int initialCount) :
        this(initialCount, string.Empty)
    {
    }
    public MyClass(int initialCount, string name)
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

C# 4.0支持默认参数，你可以将构造函数的代码进一步减少。你可以在一个构造函数中使用默认参数替换几个或者所有的参数的值：

```
public class MyClass
    {
    // collection of data
    private List<ImportantData> coll;
    // Name of the instance:
    private string name;
    // Needed to satisfy the new() constraint.
    public MyClass() :
        this(0, string.Empty)
    {
    }
    public MyClass(int initialCount = 0, string name = "")
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

选择默认参数而不是多个函数重载是一个很好的权衡（查看原则10）。默认参数可以有更多选择。上面的 MyClass 就指定了两个参数的默认值。使用者可以指定每个参数的值。使用重载产生所有排列构造函数需要四个不同的重载构造函数：没有参数的构造函数，需要参数为 initial 的构造函数，需要参数为 name 的构造函数，以及两个参数都需要的构造函数。你的类的成员变量个数增加，重载构造函数的个数就以所有参数的排列数增长。这么复杂使得默认参数是一个非常有效的方法减少潜在需要重载构造函数的数量。

定义所有参数默认参数的构造函数意味着你代码 调用 new MyClass() 是合法的。当你希望这个概念，你需要显示创建一个无参的构造函数，正如上面例子一样。然而大部分的有默认参数，泛型的类使用 new() 约束不会接受使用所有默认参数的构造函数。为了满足 new() 约束，类必须有一个显示无参构造函数。总之，你需要有一个无参的构造函数满足泛型类或泛型方法的 new() 的约束。这也不是说所有的类都需要一个无参构造函数。然后如果你支持这个，你可以保证无参构造函数在所有情况都可以工作，即使是 new() 约束的泛型类的调用。

你该注意到第二个构造函数指定 "" 为 name 参数的默认值，而不是更常见的 string.Empty 。因为 string.Empty 不是编译时期常量。它是 string 类定义的 static 属性。因为他不是编译时期的常量，你不能使用它作为参数的默认值。

然而，使用默认参数代替函数重载使得你的类和所有使用它的客户更加耦合（查看原则10）。特别地，参数的名字和默认值变成了 public 接口的一部分。改变参数的默认值需要重新编译客户的代码才能发现这些改变。这使得重载构造函数在潜在变化的未来有更多弹性。你可以添加新的构造函数，或者改变那些没有指定值而不会破坏客户端代码的构造函数的默认行为。

C# 版本1到3是不支持默认参数，尽管这是这个问题更好的解决方法。你必须使得每个构造函数就像单独的函数。随着构造函数的增多，意味着会有更多的重复代码。使用构造函数链，即让一个构造函数调用另外一个构造函数，而不是创建通用的方法。一些低效就会在提取构造函数通用逻辑的替代方法中体现出来：

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Name of the instance:
    private string name;
    public MyClass()
    {
        commonConstructor(0, "");
    }
    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, "");
    }
    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }
    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
            new List<ImportantData>(count) :
            new List<ImportantData>();
        this.name = name;
    }
}
```

这个版本看起效果一样，但是它产生太多低效的对象代码。编译器会增加几个函数代码在你的构造函数中。它会增加对所有成员变量的进行初始化。它会调用基类的构造函数。如果你写了自己的通用函数，编译器不会提取重复代码。如果你按着下面的方式写， IL 代码和第二个版本是一样的：

```
public class MyClass
{
    private List<ImportantData> coll;
    private string name;
    public MyClass()
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(0, "");
    }
    public MyClass(int initialCount)
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(initialCount, "");
    }
    public MyClass(int initialCount, string Name)
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(initialCount, Name);
    }
    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

如果你用第一个版本写构造函数，在编译看来，你是这样写的：

```
// Not legal, illustrates IL generated:
public class MyClass
{
    private List<ImportantData> coll;
    private string name;
    public MyClass()
    {
        // No variable initializers here.
        // Call the third constructor, shown below.
        this(0, ""); // Not legal, illustrative only.
    }
    public MyClass(int initialCount)
    {
        // No variable initializers here.
        // Call the third constructor, shown below.
        this(initialCount, "");
    }
    public MyClass(int initialCount, string Name)
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        name = Name;
    }
}
```

区别在于编译器既不用重复产生基类构造函数，又不用复制实例变量初始化语法到每个构造函数中。实际上基类构造函数只会在最后一个的构造函数中才是有意义的：你不能包含多于一个构造函数的初始化。你可以使用 this() 把这个委托给另一个方法，或者使用 base() 调用基类构造函数。但你不能同时调用这两个。

如果你还不清楚构造函数的初始化语法？你可以考虑下只读常量。在这个例子里，对象的 name 这就是说，你应该设置为只读。这就会导致使用通用函数构造对象产生编译错误：

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Number for this instance
    private int counter;
    // Name of the instance:
    private readonly string name;
    public MyClass()
    {
        commonConstructor(0, string.Empty);
    }
    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, string.Empty);
    }
    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }
    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        // ERROR changing the name outside of a constructor.
        this.name = name;
    }
}
```

C++ 程序员会在每个构造函数中初始化 name ,或者在通用函数中扔掉它的常量性质。 C# 的构造函数提供了更好的选择。但在大多数琐碎的类都会包含不止一个构造函数。它们的职责就是初始化对象的所有成员变量。这些函数有相似或者相同的共享的逻辑是很常见的。使用 C# 构造函数初始化语法去提取它们通用的算法，以至于你可以只把它们写一次而且执行一次。

默认参数和重载都各有优劣。一般来说，你应该选择默认参数而不是重载构造函数。毕竟，如果你让客户端指定所有参数的值，你构造函数必须能够处理所有他们指定的值。你的原始默认值应该总是显而易见的并且不会产生异常。因此，即使改变默认参数的值是技术上的破坏，对于客户端代码是不会察觉到的。他们的代码还是用原来的值，而且那些原来的值应该能产生可以接受的行为。所以要最小化使用默认参数的潜在危害。

这是最后一个关于 C# 对象初始化的原则。是时候该回顾一个类型对象构造的一系列的事件的顺序了。你应该尽量让每个成员变量在整个构造过程中只精确的初始化一次。最好的方式是你完成变量初始化越早越好。下面就是一个实例第一次构造对象的操作顺序：

1.static 变量默认存储为0。

2.static 变量初始化执行。

3.基类静态构造函数执行。

4.静态构造函数执行。

5.实例成员变量默认存储为0。

6.实例成员变量初始化执行。

7.恰当的基类实例构造函数执行。

8.实例构造函数执行。

后续同一类型的对象初始化从第5步开始因为类初始化只会执行一次。而且，步骤6和7会被优化构造函数初始化语法会引起编译器移除重复的指令。

C# 编译器保证一个对象被创建是所有东西都被初始化好。至少，在对象创建时，所有对象内存值都被设置为0是确定的。这对静态成员变量和实例成员变量都是正确的。你的目标就是确保你初始化所有变量的代码只执行一次。适用初始化语法去初始化简单的资源。适用构造函数初始化需要复杂逻辑的变量。同时提取调用其他构造函数，以最小化重复。

小结：

这个原则讲的内容还是很重要的，尤其是最后的总结——对象变量初始化的过程。总之，对于复杂静态变量逻辑（特别是会有异常抛出）使用静态构造函数实现。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2078078

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则15：使用 using 和 try/finally 清理资源

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

非托管资源类型必须使用 IDisposable 接口的 Dispose() 方法释放。.NET 的这个规则使得释放资源的职责是类型的使用者，而不是类型和系统。因此，任何时候你使用的类型有 Dispose() 方法，你就有责任调用 Dispose() 释放资源。最好的方法来保证 Dispose() 被调用时使用 using 语句或 try/finally 块。

所有包含非托管资源的类型都应该实现 IDisposable 接口。此外，如果你没有恰当的回收这些类型，它们会被动创建析构函数。如果你忘记回收这些对象，这些非内存资源会在晚些时候析构函数被执行的时候释放。这就会使得这些对象在内存待的时间更久，从而会使得应用程序因占用系统资源过多而变慢。

幸运的是， C# 语言设计者知道显示释放资源是一个很常见的操作。他们给语言添加了关键字使得会更容易。

```
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = new SqlConnection(connString);
    SqlCommand mySqlCommand = new SqlCommand(commandString, myConnection);
    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
}
```

在这个例子中，有两个可回收的对象没有恰当地被清理：SqlConnection 和 SqlCommand 。这两个对象会一直停留在内存中直到它们的析构函数被调用。（这两个类都从 System.ComponentModel.Component 继承析构函数。

通过在执行命令和连接结束的时候调用 Dispose 修复这个这个问题：

```
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = new SqlConnection(connString);
    SqlCommand mySqlCommand = new SqlCommand(commandString, myConnection);
    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
    mySqlCommand.Dispose();
    myConnection.Dispose();
}
```

那样处理就很好了，除非这个 SQL 命令执行抛出了异常。这时，上面例子的 Dispose() 就不会被执行。使用 using 语句可以保证 Dispose() 被调用。你使用 using 语句分配对象，C# 编译器就会产生 try/finally 块包含这些对象：

```
public void ExecuteCommand(string connString, string commandString)
{
    using (SqlConnection myConnection = new SqlConnection(connString))
    {
        using (SqlCommand mySqlCommand = new SqlCommand(commandString, myConnection))
        {
            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
    }
}
```

当你在函数使用一个可回收对象时， using 块是最简单方法确保对象被恰当回收。using 语句会产生 try/finally 块包裹被分配的对象。下面两段代码的 IL 代码是一样的：

```
SqlConnection myConnection = null;
// Example Using clause:
using (myConnection = new SqlConnection(connString))
{
    myConnection.Open();
}
// example Try / Catch block:
try
{
    myConnection = new SqlConnection(connString);
    myConnection.Open();
}
finally
{
    myConnection.Dispose();
}
```

如果你对没有实现 IDisposable 类型上使用 using 语句，C# 编译器会报错。例如：

```
// Does not compile:
// String is sealed, and does not support IDisposable.
using (string msg = "This is a message")
Console.WriteLine(msg);
```

using 语句对在编译时期类型实现 IDisposable 接口才能正常工作。你不能对任意对象使用：

```
// Does not compile.
// Object does not support IDisposable.
using (object obj = Factory.CreateResource())
Console.WriteLine(obj.ToString());
```

快速保护方法是使用 as 语句可以转换为安全可回收对象不管是否实现 IDisposable 接口：

```
// The correct fix.
// Object may or may not support IDisposable.
object obj = Factory.CreateResource();
using (obj as IDisposable)
Console.WriteLine(obj.ToString());
```

如果 obj 实现了 IDisposable， using 语句会产生清理的代码。否则，using 语句变为 using(null) 这样是安全的而且不做任何处理。如果你还是不确定使用 using 块包裹对象是否是正确的，为了安全：假设那样做是正确的并且像前面的方法一样使用 using 包裹对象。

这是只是介绍一个简单的情况：当你在对象内使用可回收的局部对象，使用 using 语句包裹这个对象。现在看几个复杂的用法。在第一个例子两个不同对象需要回收：连接和命令。前面的做法是使用两个 using 语句，将需要回收的两个对象分别放在里面。每个 using 语句都会产生一个 try/finally 块。等价于你写了下面的代码：

```
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = null;
    SqlCommand mySqlCommand = null;
    try
    {
        myConnection = new SqlConnection(connString);
        try
        {
            mySqlCommand = new SqlCommand(commandString, myConnection);
            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
        finally
        {
        if (mySqlCommand != null)
            mySqlCommand.Dispose();
        }
    }
    finally
    {
    if (myConnection != null)
        myConnection.Dispose();
    }
}
```

每个 using 语句都会创建一个嵌套的 try/finally 块。幸好，我们很少会在一个分配两个实现 IDisposable 不同的对象。这种情况下，可以允许那样，因为它能正常工作。但是，如果你要分配多个实现 IDisposable 的对象时，会是一个很糟糕的实现，我更喜欢自己写 try/finally 块：

```
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = null;
    SqlCommand mySqlCommand = null;
    try
    {
        myConnection = new SqlConnection(connString);
        mySqlCommand = new SqlCommand(commandString,
        myConnection);
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
        if (mySqlCommand != null)
            mySqlCommand.Dispose();
        if (myConnection != null)
            myConnection.Dispose();
    }
}
```

唯一可以抛弃这种写法的理由是你能很容易的使用 using 和 as 语句实现：

这看上去很清晰，但是会有一个很狡猾的问题。 如果 SqlCommand() 构造抛出异常 SqlConnection 对象将不会被回收。 myConnection 已经被创建，但是当 SqlCommand 构造函数执行时代码就不会进入 using 块。在 using 块没有构造好， Dispose 的调用就会被跳过。你必须确保任何实现 IDisposable 的对象的分配要在 using 块或 try 块内。否则，资源泄露就会发生。

```
public void ExecuteCommand(string connString, string commandString)
{
    // Bad idea. Potential resource leak lurks!
    SqlConnection myConnection = new SqlConnection(connString);
    SqlCommand mySqlCommand = new SqlCommand(commandString,myConnection);
    using (myConnection as IDisposable)
    using (mySqlCommand as IDisposable)
    {
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
}
```

到目前为止，你已经学会了两种最常见的情况了。当你在一个函数只分配一个可回收对象，使用 using 语句是最好方法确保资源被释放。如果在一个方法中要分配多个对象，创建多个 using 块或自己写一个 try/finally 块。

释放不同的可回收对象会有一些细微的差别。有些类型同时支持 Dispose 方法和 Close 方法去释放资源。 SqlConnection 就是其中之一。你像下面那样可以关闭 SqlConnection ：

```
public void ExecuteCommand(string connString, string commandString)
{
    SqlConnection myConnection = null;
    try
    {
        myConnection = new SqlConnection(connString);
        SqlCommand mySqlCommand = new SqlCommand
        (commandString, myConnection);
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
        if (myConnection != null)
            myConnection.Close();
    }
}
```

这个版本是关闭连接，但跟回收它还是有一些不一样。Dispose 方法不只是释放资源：它还会告诉垃圾回收器这个对象不用执行析构函数。 Dispose 会调用 GC.SuppressFinalize() 。Close 就没有这样的操作。结果，对象还待在析构队列中，即使析构已经不需要。如果你可以选择， Dispose 会比 Close 更好。你可以查看原则18了解所有细节。

Dispose() 不会将对象从内存中移除。它会触发对象释放非托管资源。这意味着你使用已经回收的对象会有问题。上面 SQLConnection 就是例子。 SQLConnection 的 Dispose() 方法断开了数据库的连接。在你回收这个连接之后，SQLConnection 还留着内存中，但是不再和数据库保持连接。所以，在任何地方都不要回收还在被引用的对象。

在某些方面，C# 的资源管理会比 C++ 更困难。你不能依赖最终的析构函数清理所有使用到的资源。但是垃圾回收机制对你再简单不过了。你使用到绝大多数类都没有实现 IDisposable 。在 .Net 框架的1500多个类，只有少于100个类实现了 IDisposable 。当你使用到其中的类，记得回收它们。你可以将这些对象放在 using 块或 try/finally 块中。无论你使用哪一种方式，确保每次所有对象总是都被恰当释放。

小结：

D.S.Qiu 一直都不知道 Dispose 的作用，并且也很疑惑 Close 和 Dispose 的区别（文末给予了解答，痛快）。但是对 Dispose 还是没有吃透，C# 执行 Dispose 的内部流程是怎么样的，都做了哪些操作？

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2078084

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则16：避免创建不需要的对象

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

垃圾回收期在管理内存方面非常出色，它非常高效地移除不再使用的对象。但是无论你如何看待它，分配和销毁一个基于堆内存的对象花费处理器时间比分配和销毁不是基于堆内存的对象要多。在函数内创建大量的引用类型对象会引入严重的性能消耗问题。

所以不能让垃圾回收器超负荷工作。你可以借鉴一些简单的技巧最小化垃圾回收器的工作。所有的引用类型对象，即使是局部变量，都被分配存储在堆内存上。每个引用类型的局部变量在函数结束都会变为垃圾。一个最常见的坏实践是在 Windows 画图处理申请 GDI 对象：

```
// Sample one
protected override void OnPaint(PaintEventArgs e)
{
    // Bad. Created the same font every paint event.
    using (Font MyFont = new Font("Arial", 10.0f))
    {
        e.Graphics.DrawString(DateTime.Now.ToString(), MyFont, Brushes.Black, new PointF(
    }
    base.OnPaint(e);
}
```

OnPainr() 会被频繁调用。每次被调用，你都会创建另一个包含相同设置的 Font 对象。垃圾回收器每次都需要为你清理。这间接导致低效。

相反，将 Font 对象从局部变量提升为成员变量。每次都重复使用相同的字体话窗口：

```
private readonly Font myFont = new Font("Arial", 10.0f);
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.DrawString(DateTime.Now.ToString(), myFont, Brushes.Black, new PointF(0, 0
}
```

你的程序不会每个绘画事件都产生垃圾。垃圾回收器的负担更少。你的程序也会跑的更快一点。当你把一个实现 IDisposable 的对象由局部变量提升为成员变量，如字体，那么你这个的类也要实现 IDisposable 。原则18会解释为什么那样做是对的。

如果一个引用类型的局部变量在函数中使用非常频繁，你需要将它提升为成员变量（值类型无关紧要）。上面绘画的字体就是最好的例子。只有常用的局部变量要频繁被访问才是好的候选。不是频繁调用就不用了。你应该避免重复创建相同的对象，也不要将每个局部变量转换为成员变量。

前面例子的静态属性 Brushes.Black 又是一个演示避免重复创建相同对象的技术。将常用的对象实例创建为静态成员变量。考虑前面例子中的黑色画刷。每次在要窗口使用黑色画东西，你都需要黑色画刷。如果你每次需要画东西都是申请一个新的，你会在运行中创建和销毁大量的黑色画刷。第一种做法是在你类中创建一个黑色画刷的成员变量，但这是远远不够的。程序可能创建大量窗口和控制，就会创建大量的黑色画刷。 .NET 框架设计者考虑到这点并且只创建一个黑色画刷让你需要的时候重复使用。 Brushes 类包含了大量的静态 Brush 对象，每个都是一种的常见颜色。在内部， Brushes 类使用懒惰算法，即只有当你需要的时候才创建。下面就是一个简单的实现：

```
private static Brush blackBrush;
public static Brush Black
{
    get
    {
        if (blackBrush == null)
            blackBrush = new SolidBrush(Color.Black);
        return blackBrush;
    }
}
```

当你第一次使用黑色画刷， Brushes 类就会创建它。 Brushes 类保持黑色画刷的一个简单的引用，当你再需要的时就会返回这个引用。结果就是你只创建了一个黑色画刷并且可以一直重复使用。并且，如果你程序不需要穿件一个特殊的资源——比如，柠檬绿画刷——它就不会被创建。框架提供方法限制了对象的创建，在你完成目标的情况下使用最少的对象集。学会在你的程序中使用这样的技巧。

你已经学会两种技巧减少申请对象的数量，就像完成自己的业务一样。你可以将经常使用的局部变量提升为成员变量。你也可以使用单例模式让一个类提供常用的实例。最后这种技巧还包含了不可变类型的最终值。 System.String 类是不可变的：你构建好一个字符串之后，这个字符串的内容就不会被修改。任何时候你写的代码看起来像修改了字符串的内容，其实是创建了一个新的字符串对象，而旧的字符串就变为垃圾。这看似很无辜的实现：

```
string msg = "Hello, ";
msg += thisUser.Name;
msg += ". Today is ";
msg += System.DateTime.Now.ToString();
```

和下面的写法一样是低效的：

```
string msg = "Hello, ";
// Not legal, for illustration only:
string tmp1 = new String(msg + thisUser.Name);
msg = tmp1; // "Hello " is garbage.
string tmp2 = new String(msg + ". Today is ");
msg = tmp2; // "Hello <user>" is garbage.
string tmp3 = new String(msg + DateTime.Now.ToString());
msg = tmp3; // "Hello <user>. Today is " is garbage.
```

字符串 tmp1, tmp2,和 tmp3 以及开始创建的消息（“Hello”），都是垃圾。string 类的 += 方法会创建新的字符串并返回。在字符串上连接字符不会修改原来的字符串。像前面的构造，可以使用 string.Format() 方法：

```
string msg = string.Format("Hello, {0}. Today is {1}",thisUser.Name, DateTime.Now.ToStrin
```

对于更复杂的字符串操作，你可以使用 StringBuilder 类：

```
StringBuilder msg = new StringBuilder("Hello, ");
msg.Append(thisUser.Name);
msg.Append(". Today is ");
msg.Append(DateTime.Now.ToString());
string finalMsg = msg.ToString();
```

StringBuilder 是一个可变字符串类用于构建不可变的 string 对象。在形成不可变 string 对象之前，它提供了配套的可变字符串方法去创建和修改文本内容。使用 StringBuilder 创建最终版本的字符串对象。更重要的是，学会这种设计思想。考虑当你设计不可变类型时（查看原则20），考虑创建一个对象去构建需要多次构造而成的最终对象。可以让使用者一步一步构建对象，并且维护这个不可变的类型。

垃圾回收器高效地管理你程序使用的内存。但是记住创建和销毁堆内存对象仍会花费时间。避免创建大量对象，不要创建你不需要的对象。同时避免创建在函数内部创建多个引用类型对象。相反，考虑提升局部变量为成员变量，或者为你的类型创建大多数常用的实例。最后，考虑创建一个可变类来构建不可变类。

小结：

这则原则相对简短，但是非常重要，尤其是对于菜鸟码农，总之，避免重复创建对象。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2078554

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则17：实现标准的 Dispose 模式

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

我们已经讨论过回收持有非托管资源对象的重要性。现在就介绍怎么实现管理类所包含不是内存的资源的代码。 .NET 框架回收非托管资源已经有了标准的模式。你的类的使用者会希望你遵循这个标准模式。标准的回收习惯是使用者调用你实现的 IDisposable 接口，如果使用者忘记了析构函数也会被动执行。它和垃圾回收器一起工作，保证你的对象在必要的时候只受到析构函数带来的性能损失。这就是正确处理非托管资源的方式，所以你需要完全弄明白它。

在继承层次的基类应该实现 IDisposable 接口以释放资源。这个类也应添加析构函数作为被动的调用。这两个方法都是通过虚函数实现资源释放，以至于子类可以根据它们自己的资源管理需求进行重载。子类只有当需要释放自己的资源的时候才需要重载虚函数，并且记得调用基类的实现版本。

开始时，如果你的类使用非托管资源，这个类就必须有析构函数。你不应该依赖使用者总是调用 Dispose() 方法。如果他们忘记了你就会出现内存泄露问题。没有调用 Dispose 是他们的过失，但是你也会受到他们的责备。唯一的保证非托管资源会被正确的释放的方式是创建析构函数。所以就创建一个。

当垃圾回收器运行时，它会立即把没有析构函数的对象从内存中移除。有析构函数的所有对象还会留在内存中。这些对象会添加到析构执行队列中，并且垃圾回收器起一个新线程执行这些对象析构函数。析构线程完成工作后，垃圾对象才会从内存中移除。需要析构的对象在内存中停留的时间会比没有析构函数的对象久点。但是你别无选择。如果你的类持有非托管资源，你就得被动实现析构函数。但是你不用太担心性能。下一步保证用户使用更简单，而且可以避免析构函数造成的性能损失。

实现 IDisposable 是标准的方式告诉用户和系统你的对象持有必须被及时释放的资源。IDiposable 接口只包含一个方法：

```
public interface IDisposable
{
    void Dispose();
}
```

实现 IDisposable.Dispose() 方法要负责四个任务：

1.释放所有非托管资源。

2.释放所有托管资源（包括卸载事件）。

3.设置表示对象依据被清理的标记位。在 public 方法你需要坚持这个状态值，并且如果已经被回收对象的调用要抛出 ObjectDisposed 异常。

4.阻止析构。你通过调用 GC.SuppressFinalize(this) 来完成这项工作。

实现 IDisposable ，你完成两件事情：第一个你提供了一种及时释放托管资源的机制，另一个是你提供了标准模式让用户释放非托管资源。这是非常重要的。你为类型实现 IDisposable 后，用户可以避免析构的开销。你的类在 .NET 环境中就理所当然有很好的表现。

但是这个机制还是会有些坑。怎么样做到子类清理自己的资源并且基类同时也进行清理。如果子类重载了析构函数或者添加自己的 IDisposable 实现，这两个方法必须调用基类的对应的实现。否则，基类就不会被正确的清理。并且，析构函数和 Dispose 有着相同的职责：几乎可以肯定析构函数和 Dispose 方法中会有重复的代码。后面你会在原则23中学到，重载接口函数不会像你期待那样工作。标准模式中的第三个方法，通过一个受保护的辅助性虚函数，提取出它们的常规任务并且让子类来释放自己的资源。基类包含接口的核心代码， 子类提供的 Dispose()虚函数或者析构函数来负责清理资源：

```
protected virtual void Dispose(bool isDisposing)
```

方法重载这个方法是非常必须的以同时支持析构函数和 Dispose，并且因为它虚函数，可以为所有子类提供入口。子类可以重载这个方法，提供恰当的实现去清理它们的资源，和调用基类的实现。当 isDisposing 为 true 时，你清理托管资源和非托管资源，如果为 false ,你只清理了非托管资源。这两种情况，都要调用基类的 Dispose(bool) 方法去清理基类的资源。

下面这个简单例子为你展示如何实现这个模式。 MyResourceHog 类演示实现了 IDisposable 和创建一个虚 Dispose 方法：

```
public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    private bool alreadyDisposed = false;
    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    // Virtual Dispose method
    protected virtual void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (alreadyDisposed)
        return;
        if (isDisposing)
        {
        // elided: free managed resources here.
        }
        // elided: free unmanaged resources here.
        // Set disposed flag:
        alreadyDisposed = true;
    }
    public void ExampleMethod()
    {
        if (alreadyDisposed)
            throw new ObjectDisposedException("MyResourceHog","Called Example Method on D
        // remainder elided.
    }
}
```

如果你一个子类需要额外的清理，可以实现 protected Dispose 方法：

```
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool disposed = false;
    protected override void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (disposed)
        return;
        if (isDisposing)
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.
        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);
        // Set derived class disposed flag:
        disposed = true;
    }
}
```

注意到基类和子类都包含了一个回收状态的标记位。这完全是没有必要的。重复的标记可以掩盖在回收时所有可能出现的错误，标记位是相对对象的每个类型而言的，而不是所有类型。

你要被动地写 Dispose 和析构函数。对象的回收顺序可以是任意的。你可能会碰到在对象被回收之前，它的成员变量之一已经被回收了的情况。你可能还没发现这个问题因为 Dispose() 方法可被执行多次。如果对象已经被回收被再次调用，就什么也不做。析构函数的规则也一样。任何对象的引用还在内存，就不用检查 null 引用。然而任何你引用的对象都可能被回收了。或者已经被析构了。

你还会主要到 MyResourceHog 和 DerivedResourceHog 都没有包含析构函数。这个例子的代码没有直接包含任何非托管资源。所以，析构函数是不需要的。这意味着这里代码不需要调用 Dispose(false) 。这是就是正确的模式。除非你的类直接包含了非托管资源，否则你不需要析构函数。只有那些直接包含非托管资源的类才需要实现析构函数并且增加开销。即使没有被调用，析构函数的出现都会引入相当大的性能损失。除非你的类型需要析构函数，否则就不要添加它。然后，你仍要正确实现这个模式以保证只要子类添加非托管资源，就要添加析构函数，并且实现 Dispose(bool) 能正确处理非托管资源。

这带给我最重要的建议就是任何涉及回收或清理的方法：你都应该只是释放资源。不要在回收清理方法做其他操作。因为在 Dispose 和析构函数中处理其他操作会引入严重的问题。对象由构造函数而生，由垃圾回收器回收而灭亡。当你程序没有访问它们，你可以把它们当做休眠状态。如果你不能访问对象，就不能调用它的方法。对于这样的意图和目的，它就是死的。但是有析构函数的对象在它们被宣告死亡之前还能喘最后一口气。析构函数应该只是清理非托管资源。如果在析构函数访问这个对象，那么这个对象又会复活了。它还是活的但是不好，即使从休眠状态唤醒了。下面就是很明显的例子：

```
public class BadClass
{
    // Store a reference to a global object:
    private static readonly List<BadClass> finalizedList =
    new List<BadClass>();
    private string msg;
    public BadClass(string msg)
    {
        // cache the reference:
        msg = (string)msg.Clone();
    }
    ~BadClass()
    {
        // Add this object to the list.
        // This object is reachable, no
        // longer garbage. It's Back!
        finalizedList.Add(this);
    }
}
```

当 BadClass 对象执行析构函数式，它把自己的引用添加到队列里去。这又使得它自己可被访问。也就是它还活着！你所引入的对象问题是任何人都畏缩的。对象一旦析构了，垃圾回收器就会相信它不再需要调用析构函数。如果你实际想在析构函数中复活一个对象，这是不会发生的。第二，你的一些资源可能不能再被访问。GC 将不会从内存移除任何只有析构队列能访问的对象。但是它可能已经被析构过了。如果是这样，它们绝大多数不能再被使用。尽管 BadClass 拥有的成员还是停留在内存，它们看起来很像被回收或析构了。你没有任何方法能控制析构的顺序。你不能使这种结构的工作可靠。情不要尝试。

除了学院练习，我从来没有看过像这样复活对象的代码。但我见过析构函数试图做这件事调用函数保持引用带回到有生命状态。原则上析构函数上的任何代码都要非常小心，推而广之， Dispose 方法也一样。如果代码除了释放资源之外还做了其他任何事情，请再次确认。这次操作很可能在将来发生问题。移除这些操作，并且保证析构函数和 Dispose() 函数只是释放资源。

在托管的环境，你不需要为每个类都写一个析构函数；只有存储了非托管类型或者包含实现 IDisposable 的成员变量的类才需要。即使你只是需要 Disposable 接口，而不需要析构函数，但实现整个模式。否则，你限制你的子类以至于它们实现标准 Dispose 模式会很复杂。遵循我描述的标准 Dispose 模式习惯。那对于你，你类的使用者，继承你类的人都会是更简单的。

小结：

这篇介绍实现 Dispose 接口的准则，对于有引用非托管资源的就要实现 Dispose 接口并且写析构函数，具体的实现模式文中有详实的介绍。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2078979

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则17：实现标准的 Dispose 模式

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

我们已经讨论过回收持有非托管资源对象的重要性。现在就介绍怎么实现管理类所包含不是内存的资源的代码。 .NET 框架回收非托管资源已经有了标准的模式。你的类的使用者会希望你遵循这个标准模式。标准的回收习惯是使用者调用你实现的 IDisposable 接口，如果使用者忘记了析构函数也会被动执行。它和垃圾回收器一起工作，保证你的对象在必要的时候只受到析构函数带来的性能损失。这就是正确处理非托管资源的方式，所以你需要完全弄明白它。

在继承层次的基类应该实现 IDisposable 接口以释放资源。这个类也应添加析构函数作为被动的调用。这两个方法都是通过虚函数实现资源释放，以至于子类可以根据它们自己的资源管理需求进行重载。子类只有当需要释放自己的资源的时候才需要重载虚函数，并且记得调用基类的实现版本。

开始时，如果你的类使用非托管资源，这个类就必须有析构函数。你不应该依赖使用者总是调用 Dispose() 方法。如果他们忘记了你就会出现内存泄露问题。没有调用 Dispose 是他们的过失，但是你也会受到他们的责备。唯一的保证非托管资源会被正确的释放的方式是创建析构函数。所以就创建一个。

当垃圾回收器运行时，它会立即把没有析构函数的对象从内存中移除。有析构函数的所有对象还会留在内存中。这些对象会添加到析构执行队列中，并且垃圾回收器起一个新线程执行这些对象析构函数。析构线程完成工作后，垃圾对象才会从内存中移除。需要析构的对象在内存中停留的时间会比没有析构函数的对象久点。但是你别无选择。如果你的类持有非托管资源，你就得被动实现析构函数。但是你不用太担心性能。下一步保证用户使用更简单，而且可以避免析构函数造成的性能损失。

实现 IDisposable 是标准的方式告诉用户和系统你的对象持有必须被及时释放的资源。IDiposable 接口只包含一个方法：

```
public interface IDisposable
{
    void Dispose();
}
```

实现 IDisposable.Dispose() 方法要负责四个任务：

1.释放所有非托管资源。

2.释放所有托管资源（包括卸载事件）。

3.设置表示对象依据被清理的标记位。在 public 方法你需要坚持这个状态值，并且如果已经被回收对象的调用要抛出 ObjectDisposed 异常。

4.阻止析构。你通过调用 GC.SuppressFinalize(this) 来完成这项工作。

实现 IDisposable，你完成两件事情：第一个你提供了一种及时释放托管资源的机制，另一个是你提供了标准模式让用户释放非托管资源。这是非常重要的。你为类型实现 IDisposable 后，用户可以避免析构的开销。你的类在 .NET 环境中就理所当然有很好的表现。

但是这个机制还是会有些坑。怎么样做到子类清理自己的资源并且基类同时也进行清理。如果子类重载了析构函数或者添加自己的 IDisposable 实现，这两个方法必须调用基类的对应的实现。否则，基类就不会被正确的清理。并且，析构函数和 Dispose 有着相同的职责：几乎可以肯定析构函数和 Dispose 方法中会有重复的代码。后面你会在原则23中学到，重载接口函数不会像你期待那样工作。标准模式中的第三个方法，通过一个受保护的辅助性虚函数，提取出它们的常规任务并且让子类来释放自己的资源。基类包含接口的核心代码，子类提供的 Dispose()虚函数或者析构函数来负责清理资源：

```
protected virtual void Dispose(bool isDisposing)
```

方法重载这个方法是非常必须的以同时支持析构函数和 Dispose，并且因为它虚函数，可以为所有子类提供入口。子类可以重载这个方法，提供恰当的实现去清理它们的资源，和调用基类的实现。当 isDisposing 为 true 时，你清理托管资源和非托管资源，如果为 false ,你只清理了非托管资源。这两种情况，都要调用基类的 Dispose(bool) 方法去清理基类的资源。

下面这个简单例子为你展示如何实现这个模式。 MyResourceHog 类演示实现了 IDisposable 和创建一个虚 Dispose 方法：

```
public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    private bool alreadyDisposed = false;
    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    // Virtual Dispose method
    protected virtual void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (alreadyDisposed)
        return;
        if (isDisposing)
        {
        // elided: free managed resources here.
        }
        // elided: free unmanaged resources here.
        // Set disposed flag:
        alreadyDisposed = true;
    }
    public void ExampleMethod()
    {
        if (alreadyDisposed)
            throw new ObjectDisposedException("MyResourceHog","Called Example Method on D
        // remainder elided.
    }
}
```

如果你一个子类需要额外的清理，可以实现 protected Dispose 方法：

```
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool disposed = false;
    protected override void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (disposed)
        return;
        if (isDisposing)
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.
        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);
        // Set derived class disposed flag:
        disposed = true;
    }
}
```

注意到基类和子类都包含了一个回收状态的标记位。这完全是没有必要的。重复的标记可以掩盖在回收时所有可能出现的错误，标记位是相对对象的每个类型而言的，而不是所有类型。

你要被动地写 Dispose 和析构函数。对象的回收顺序可以是任意的。你可能会碰到在对象被回收之前，它的成员变量之一已经被回收了的情况。你可能还没发现这个问题因为 Dispose() 方法可被执行多次。如果对象已经被回收被再次调用，就什么也不做。析构函数的规则也一样。任何对象的引用还在内存，就不用检查 null 引用。然而任何你引用的对象都可能被回收了。或者已经被析构了。

你还会主要到 MyResourceHog 和 DerivedResourceHog 都没有包含析构函数。这个例子的代码没有直接包含任何非托管资源。所以，析构函数是不需要的。这意味着这里代码不需要调用 Dispose(false) 。这是就是正确的模式。除非你的类直接包含了非托管资源，否则你不需要析构函数。只有那些直接包含非托管资源的类才需要实现析构函数并且增加开销。即使没有被调用，析构函数的出现都会引入相当大的性能损失。除非你的类型需要析构函数，否则就不要添加它。然后，你仍要正确实现这个模式以保证只要子类添加非托管资源，就要添加析构函数，并且实现 Dispose(bool) 能正确处理非托管资源。

这带给我最重要的建议就是任何涉及回收或清理的方法：你都应该只是释放资源。不要在回收清理方法做其他操作。因为在 Dispose 和析构函数中处理其他操作会引入严重的问题。对象由构造函数而生，由垃圾回收器回收而灭亡。当你程序没有访问它们，你可以把它们当做休眠状态。如果你不能访问对象，就不能调用它的方法。对于这样的意图和目的，它就是死的。但是有析构函数的对象在它们被宣告死亡之前还能喘最后一口气。析构函数应该只是清理非托管资源。如果在析构函数访问这个对象，那么这个对象又会复活了。它还是活的但是不好，即使从休眠状态唤醒了。下面就是很明显的例子：

```
public class BadClass
{
    // Store a reference to a global object:
    private static readonly List<BadClass> finalizedList =
    new List<BadClass>();
    private string msg;
    public BadClass(string msg)
    {
        // cache the reference:
        msg = (string)msg.Clone();
    }
    ~BadClass()
    {
        // Add this object to the list.
        // This object is reachable, no
        // longer garbage. It's Back!
        finalizedList.Add(this);
    }
}
```

当 BadClass 对象执行析构函数式，它把自己的引用添加到队列里去。这又使得它自己可被访问。也就是它还活着！你所引入的对象问题是任何人都畏缩的。对象一旦析构了，垃圾回收器就会相信它不再需要调用析构函数。如果你实际想在析构函数中复活一个对象，这是不会发生的。第二，你的一些资源可能不能再被访问。GC 将不会从内存移除任何只有析构队列能访问的对象。但是它可能已经被析构过了。如果是这样，它们绝大多数不能再被使用。尽管 BadClass 拥有的成员还是停留在内存，它们看起来很像被回收或析构了。你没有任何方法能控制析构的顺序。你不能使这种结构的工作可靠。情不要尝试。

除了学院练习，我从来没有看过像这样复活对象的代码。但我见过析构函数试图做这件事调用函数保持引用带回到有生命状态。原则上析构函数上的任何代码都要非常小心，推而广之， Dispose 方法也一样。如果代码除了释放资源之外还做了其他任何事情，请再次确认。这次操作很可能在将来发生问题。移除这些操作，并且保证析构函数和 Dispose() 函数只是释放资源。

在托管的环境，你不需要为每个类都写一个析构函数；只有存储了非托管类型或者包含实现 IDisposable 的成员变量的类才需要。即使你只是需要 Disposable 接口，而不需要析构函数，但实现整个模式。否则，你限制你的子类以至于它们实现标准 Dispose 模式会很复杂。遵循我描述的标准 Dispose 模式习惯。那对于你，你类的使用者，继承你类的人都会是更简单的。

小结：

这篇介绍实现 Dispose 接口的准则，对于有引用非托管资源的就要实现 Dispose 接口并且写析构函数，具体的实现模式文中有详实的介绍。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2078979

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则18：值类型和引用类型的区别

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

值类型或引用类型？结构体或类？什么时候你需要使用它们？这不是 C++， 定义的类型为值类型可以当做引用类型使用。这也不是 Java， 所有类都是引用类型（除非你是语言设计者之一）。当你创建类的时候你就需要决定这个类所有实例的行为。在开始的时候就要做好这个重要的选择。你必须面对这个选择的后果因为改变之前的选择会引起一些代码的破坏。创建类型的时候只是很简单的选择 struct 和 class 关键字，但是如果改变类型就要花很大功夫去更新客户代码。

这不想更喜欢其中的一个那么简单。正确的选择取决于你希望怎么样使用新类型。值类型不会有多态性。它们更适合用于存储程序操作的数据。引用类型具有多态性应该用来定义程序的行为。考虑新类型期待的职责，并且选择正确的类型来创建。结构体存储数据。类定义行为。

.NET 和 C# 会加入值类型和引用类型的区别是因为在 C++ 和 Java 都出现很常见的问题。在 C++ 中，所有参数和返回值都按值传递。按值传递是非常高效的，但是会有一个问题：部分复制（有时也被称作对象切割）。如果你在希望是基类的地方使用的是子类，只有子类中基类中的部分会被复制。你因此失去子类的所有信息。即使是调用虚函数也是执行的基类的版本。

Java 语言的针对这个问题处理是或多或少移除值类型。所有用户定义的类型都是引用类型。在 Java 语言中，所有参数和返回值都是按引用传递的。这个策略有着保持一致的优点，但是牺牲了性能。我们来正视这个问题，有些类不需要多态性——它们也不会被那样设计。Java 程序员为每个变量都需要堆内存的分配和最后的垃圾回收。他们还需要花费更多的时间消耗去解引用每个变量。所有变量都是引用。在 C# 中，你声明新类型是值类型还是引用类型取决于你使用的是 struct 还是 class 关键字。值类型会是小的，轻量的类型。引用类型会出现你类型的继承结构中。这个部分会检查类型的不同使用让你明白值类型和引用类型的所有区别。

首先，这个类型作为方法的返回值：

```
private MyData myData;
public MyData Foo()
{
    return myData;
}
// call it:
MyData v = Foo();
TotalSum += v.Value;
```

如果 MyData 是值类型，返回值会被复制存储为 v 。然而，如果 MyData 是引用类型，会导出把内部的变量的引用。你就破坏了封装的原则（查看原则26）。

或者，考虑下面的变种：

```
public MyData Foo2()
{
    return myData.CreateCopy();
}
// call it:
MyData v = Foo();
TotalSum += v.Value;
```

现在， v 是 myData 的复制。如果是引用类型，在堆内存会创建两个对象。你不会有暴露内部数据的问题。而是，你在堆内存创建另一个对象。如果 v 是一个局部变量，它会很快变为垃圾并且克隆会强制进行类型检查。这些都是低效率的。

如果类使用 public 方法和属性导出数据应该使用值类型。但那不是说所有有 public 的成员的类型都得是值类型。前面的代码就是假设 MyData 是存储值。它的职责就是存储这些值。

但是考虑下面代码：

```
private MyType myType;
public IMyInterface Foo3()
{
    return myType as IMyInterface;
}
// call it:
IMyInterface iMe = Foo3();
iMe.DoWork();
```

myType 仍然从 Foo3 方法中返回。但是这次，不是从返回值中访问数据，而是访问对象定义接口的方法。你访问 MyType 对象不是它存储的数据而是它的行为。那个行为是通过 IMyInferface 体现的，它可以被多个类实现。对于这里例子， MyType 就需要是引用类型，而不是值类型。 MyType 的职责是围绕它的行为，而不是它的数据。

这段简单的代码开始告诉你的区别：值类型存储值，引用类型定义的行为。现在看得更深一点，在这些类型存储在内存和存储模型上表现的性能。考虑这个类：

```
public class C
{
    private MyType a = new MyType();
    private MyType b = new MyType();
    // Remaining implementation removed.
}
C cThing = new C();
```

多个对象被创建？它们会占多大内存？这还不能下定论。如果 MyType 是值类型，你只有一次内存分配。分配的大小是 MyType 大小的两倍。然而，如果 MyType 是一个引用类型，你就有三次内存分配，第一个是 C 对象，它是一个8字节（指针占32位）， 另外两个是 C 对象

包含的 MyType 对象。这个区别是因为值类型会存储在对象内部，而引用类型不会。每个引用变量持有引用，存储需要额外的内存分配。

为了证明这一点，考虑下面内存分配：

```
MyType[] arrayOfTypes = new MyType[100];
```

如果 MyType 是值类型，会一次分配100个的 MyType 对象。然而，如果 MyType 是引用类型，只有一次分配发生。数组的每个元素都是 null 。当你初始化数组100个元素，你就已经有101内存分配——101次多一次内存分配。分配大量的引用类型会将对内存碎片化就会变得慢下来。如果你创建的类型是为了存储数据，那么值类型就是要选择的方式。

值类型和引用类型是一个很重要的选择。将值类型改为引用类型会有影响深远的改变。考虑这个类型：

```
public struct Employee
{
    // Properties elided
    public string Position
    {
        get;
        set;
    }
    public decimal CurrentPayAmount
    {
        get;
        set;
    }
    public void Pay(BankAccount b)
    {
        b.Balance += CurrentPayAmount;
    }
}
```

这个简单类型只包含一个方法让你支付被雇佣者。时间久了，系统也会运行很好。如果你决定有不同类型的被雇佣者：销售人员得到佣金，经理收到奖金。你决定改变 Employee 的类型：

```
public class Employee2
{
    // Properties elided
    public string Position
    {
        get;
        set;
    }
    public decimal CurrentPayAmount
    {
        get;
        set;
    }
    public virtual void Pay(BankAccount b)
    {
        b.Balance += CurrentPayAmount;
    }
}
```

这个会破坏很多已经存在使用你定义 struct 的代码。返回值类型变为返回引用类型。参数按值传递变为按引用传递。下面代码的行为会有巨大的变化：

```
Employee e1 = Employees.Find(e => e.Position == "CEO");
BankAccount CEOBankAccount = new BankAccount();
decimal Bonus = 10000;
e1.CurrentPayAmount += Bonus; // Add one time bonus.
e1.Pay(CEOBankAccount);
```

一次性的奖金变成了永久的加薪。这就是用引用类型替换值类型导致的。编译器会十分欢快是你的改变生效。 这个 CEO 也会很高兴。另一个头， CFO 就会报错。你不能想当然把值类型改为引用类型因为实际上：它改变行为了。

这个问题出现是因为 Employee 类不再遵循值类型的规范。定义的被雇佣者除了存储数据，在这个例子中，它还有其他职责——支付被雇佣者。职责是类类型的范畴。类可以定义多态很容易实现常见的职责，结构体就不能只应该限制存储数据。

.NET 文档推荐你考虑类型大小作为决定是否值类型和引用类型的决定因素。实际上，更好的因素是类型的使用。如果类型只是一个结构或者数据的载体最佳的候选是值类型。的确值类型在内存管理更高效：少量的堆内存碎片，更少的垃圾，以及更少寻址。更重要的是，值类型作为方法或属性的返回值会被复制。暴露内部结构体的引用没有任何危害。但是注意其他的细节。值类型对支持常见的面向对象技术很有限制。你不能创建值类型的继承结果。你应该认为所有的值类型都是封闭的。你创建的值类型可以实现接口但是需要装箱，原则17介绍拆箱操作引起的性能问题（译者注：这应该是第一步叙述，第一个版中原则17关于封箱和拆箱）。你应该认为值类型作为存储的容器，而不是面向对象意义上的对象。

你会创建比值类型更多的引用类型。如果下面的问题你都回答是，那你可以创建值类型。以前面雇佣者为例考虑这些问题：

1.这个类是否满足存储数据的职责？

2.是否所有属性 public 接口都只是访问数据？

3.我是否确定这个类型不会有子类？

4.我是否确定这个类型不会看成多态性？

构建低级的数据存储类就用值类型。使用引用类型构建你程序的行为。你可以安全的复制数在对象外暴露。你可以享受基于栈和内置值存储的内存使用优势，而且你可以利用标准的面向对象计算实现你程序的逻辑。当对类型选择很疑惑的时候，就使用引用类型。

小结：

总结下：值类型和引用类型的选择——如果只是数据的存储，并且所有 public 的接口（属性和方法）都是只是访问数据而不是修改数据才使用值类型，其他情况都选择引用类型。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2079672

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则19：确保0是值类型的一个有效状态

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

.NET 系统会将所有对象默认初始化为0。也没有方法阻止其他程序创建值类型对象并初始化为0。请使你的类型保留默认值。

有一个特例是枚举。创建不包含0的枚举是可为之的。所有的枚举都继承自 System.ValueType 。枚举的起始值是0，但是你可以修改这个行为：

```
public enum Planet
{
    // Explicitly assign values.
    // Default starts at 0 otherwise.
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Neptune = 7,
    Uranus = 8
    // First edition included Pluto.
}

Planet sphere = new Planet();
```

sphere 等于0，这不是有效值。任何代码都依赖于（正常）的事实，枚举类型都限定于定义的枚举集，所以就不能工作。如果你创建一个枚举类，确保0是其中一个值。如果你在你的枚举使用位模式，将0定义为所有其他属性都缺失。

既然如此，你可以强迫所有使用者都显示初始化值：

```
Planet sphere2 = Planet.Mars;
```

但很难构建包含这个类型的其他类型：

```
public struct ObservationData
{
    private Planet whichPlanet; //what am I looking at?
    private double magnitude; // perceived brightness.
}
```

使用者创建 ObservationData 对象会创建无效的 Planet 域：

```
ObservationData d = new ObservationData();
```

新创建的 ObservationData 有0的 magnitude，这是合理的。但是 wihchPlanet 是无效的。你需要使0是一个合理的值。如果可能，最好选择0为作为默认值。 Planet 枚举没有任何明显的默认值。当使用者没有指定默认值，它不会随意选择一个枚举值。如果你遇到这种情况，使用0作为未初始状态然后在后面更新：

```
public enum Planet2
{
    None = 0,
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Neptune = 7,
    Uranus = 8
}

Planet sphere = new Planet();
```

sphere 现在的值是空。给 Planet 枚举加上未初始化的默认值会扩散到 ObservationData 结构体中。这样新创建的 ObservationData 有一个0 magnitude 和空为目标。添加显示构造函数让使用者显示初始化所有域的值：

```
public struct ObservationData
{
    Planet whichPlanet; //what am I looking at?
    double magnitude; // perceived brightness.
    ObservationData(Planet target,
    double mag)
    {
        whichPlanet = target;
        magnitude = mag;
    }
}
```

但是记住默认构造函数仍可见的还是构造函数的一部分。使用者还是可以创建系统初始化的变量，并且你不能阻止他们。

这仍然有些错误，因为没有真的观察是没有任何意义。你可以通过改变 Observation 为类解决这个特例，这样无参构造函数就不可用了。但是，如果你创建枚举，你不可能强制其他开发者遵从这个规则。最好的最好是创建的枚举类型的0位模式是有效的，即使这不是一个完美的抽象。

在讨论其他类型之前，你需要理解使用 enum 作为标记的一些特殊例子。 enum 使用标记特性需要总是设置 None 值为0。

```
[Flags]
public enum Styles
{
    None = 0,
    Flat = 1,
    Sunken = 2,
    Raised = 4,
}
```

很多开发者对标记位枚举值使用按位运算 AND 操作符。0值在位标记会引起严重问题。下面的测试当 Flat 是0值时不会工作：

```
if ((flag & Styles.Flat) != 0) // Never true if Flat == 0\.
    DoFlatThings();
```

如果你使用标记位，确保0是有效的，它表示“没有任何标记”。

另一个常见初始化问题涉及到值类型包含引用类型。string 是常见的例子：

```
public struct LogMessage
{
    private int ErrLevel;
    private string msg;
}

LogMessage MyMessage = new LogMessage();
```

MyMessage 包含为 null 的 msg 域引用。没有任何方式去强制不同的初始化，但是你可以使用属性本地化这个问题。你可以创建一个属性想所有使用者暴露 msg 的值。添加逻辑使得属性返回空字符串而不是 null:

```
public struct LogMessage2
{
    private int ErrLevel;
    private string msg;
    public string Message
    {
        get
        {
            return (msg != null) ? msg : string.Empty;
        }
        set
        {
            msg = value;
        }
    }
}
```

你应该在你的类的内部使用这个属性。总是只在一个地方检查 null 引用。当你从你的程序集中调用 Mesaage 的访问也是 inline 的。你会活得高效的代码和最少的错误。

系统会默认初始化所有实例值为0。没有任何方法阻止使用者创建值类型的实例不都是0。如果可能，使得所有0都是自然默认值。一个特殊情况是，enum 作为标记位使用应该确保0表示没有任何标记位。

小结：

主要是 struct 和 enum 值类型的默认初始化，要能保证0值时一个有效的值。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com) 交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2079730

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则20：更倾向于使用不可变原子值类型

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

不可变类型是很简单的：一旦被创建，它们就是常量。如果你验证构造对象参数，你知道从那以后它们就是有效的状态。你不可能改变对象的内部状态让它失效。一旦对象构造好，如果不允许任何状态改变，你会省去很多必须的错误的检查。不可变类型本质上是线程安全的：多个读取者可以访问相同的内容。如果内部状态不会改变，不同线程就没有机会读取到不一致的数据。不可变类型可以让你的对象安全地暴露。调用者不能修改你的对象的内部状态。不可变类型在基于哈希的集合中工作的更好。Object.GetHashCode() 的返回值是实例不变（查看原则7）；对于不可变类型这一直是正确的。

在实践中，很难让每个类型都是不可变的。你需要克隆对象去修改任何程序状态。那就是为什么推荐使用院子并且不可变的值类型。分解你的类型为自然单一的结构体实体。 Address 类型就是这样的。一个地址就是一个简单的实体，有多个相关域构成。一个域的改变很大程度上意味着改变其他域。消费者的类型不具有原子性。消费者类型通常包含很多信息：地址，名字和一个或多个电话号码。这些独立的信息都可能改变。一个消费者可能改变电话号码而没有搬家。另一个消费者可能只改变他的货她的名字。消费者对象不是原子的；它由很多不同的不可变类型构成：地址，名字，或者电话号码的集合。原子类型是单一的实体：你可以替换原子类型的整个内容。如果改变它的一个构成域可能会出现异常。

这是地址不可变类型的典型实现：

```
// Mutable Address structure.
public struct Address
{
    private string state;
    private int zipCode;
    // Rely on the default system-generated
    // constructor.
    public string Line1
    {
        get;
        set;
    }
    public string Line2
    {
        get;
        set;
    }
    public string City
    {
        get;
        set;
    }
    public string State
    {
        get { return state; }
        set
        {
            ValidateState(value);
            state = value;
        }
    }
    public int ZipCode
    {
        get { return zipCode; }
        set
        {
            ValidateZip(value);
            zipCode = value;
        }
    }
    // other details omitted.
}
// Example usage:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now.
a1.ZipCode = 48103; // State still invalid now.
a1.State = "MI"; // Now fine.
```

内部状态的改变意味着可能破坏对象的不可变性，至少它是暂时的。你更改了 City 域，你就已经使 a1 变为无效状态了。城市改变了不可能再和州或邮政编码域匹配。这段代码看起来是无害的，但是假设它是多线程程序的一部分就不会这么认为了。在城市域改变之后和州域改变之前上下文的切换可能潜在使另外一个线程看到的是不一致的数据。

好的，所以你会觉得你写的不是多线程程序。你仍然会有麻烦。想象邮政编码是无效的，就会抛出异常。你根据你的意图做了写改变，同时使得系统就变成无效状态。为了修复这个问题，你可以在地址结构体中增加内部验证码。验证码会增加相当大的规模和复杂性。为了实

现全部异常安全，你需要被动的复制改变状态一个或多个域的代码块。线程安全需要在属性 set 和 get 访问器上增加大量的线程同步检查。总之，这是一个重大的工作，随着时间的推移可能还会扩展到你新增加的特性里面。

另外，你需要将 Address 定义为 struct 类型，使它不可变。开始让所有实例域对外部使用者变为只读：

```
public struct Address2
{
    // remaining details elided
    public string Line1
    {
        get;
        private set;
    }
    public string Line2
    {
        get;
        private set;
    }
    public string City
    {
        get;
        private set;
    }
    public string State
    {
        get;
        private set;
    }
    public int ZipCode
    {
        get;
        private set;
    }
}
```

现在，你已经得到一个基于 public 接口的不可变类型。为了使它有用，你需要添加初始化 Address 结构的构造函数。 Address 结构只需要一个构造函数，指定每个域。不需要复杂构造函数，因为赋值操作足够的高效。记住默认构造函数仍然是可用的。有一个默认的地址，它的所有字符串为 null ，而且邮政编码为0：

```
public Address2(string line1, string line2, string city, string state, int zipCode) :
    this()
{
    Line1 = line1;
    Line2 = line2;
    City = city;
    ValidateState(state);
    State = state;
    ValidateZip(zipCode);
    ZipCode = zipCode;
}
```

使用不可变类型需要一个稍微不同的调用次序去改变它的状态。你穿件一个新的对象而不是修改已存在的实例：

```
// Create an address:
Address2 a2 = new Address2("111 S. Main", "", "Anytown", "IL", 61111);
// To change, re-initialize:
a2 = new Address2(a1.Line1,a1.Line2, "Ann Arbor", "MI", 48103);
```

a1 的值有两个州：一个是原来的位置在 Anytown ，或者是后面更新的位置 Ann Arbor。你不会像之前的例子修改已存在的地址导致变为无效的临时状态。这些临时状态只存在 Address 构造器的执行过程中，在外部是不可见的。一旦新的 Address 对象构造好，它的值在任何时候都是固定不变的。这是例外的安全： a1 要么是就得值要么就是新的值。如果在构造新的 Address 对象是抛出异常，旧的值 a1 还是不会改变的。

第二个 Address 类型不是严格的不可变。带有 private set 的隐式属性仍包含方法改变内部的状态。如果想要一个真实的不可变类型，你需要做更多改变。你需要改变隐式属性为显示属性，并且修改它背后的域为 readonly：

```
public struct Address3
{
    // remaining details elided
    public string Line1
    {
        get { return Line1; }
    }
    private readonly string line1;
    public string Line2
    {
        get { return line2; }
    }
    private readonly string line2;
    public string City
    {
        get { return city; }
    }
    private readonly string city;
    public string State
    {
        get { return state; }
    }
    private readonly string state;
    public int ZipCode
    {
        get { return zip; }
    }
    private readonly int zip;
    public Address3(string line1, string line2, string city, string state, int zipCode) :
        this()
    {
        this.line1 = line1;
        this.line2 = line2;
        this.city = city;
        ValidateState(state);
        this.state = state;
        ValidateZip(zipCode);
        this.zip = zipCode;
    }
}
```

为了创建不可变类型，你需要区别没有任何漏洞让使用者改变你的内部状态。值类型不支持继承，所有你不需要防御子类修改基类的域。但是你需要注意不可变类中的可变引用类的域，你需要防御型复制可变类型。这个例子假设 Phone 是不可变的值类型因为我们只关心值类型的域的不可变性：

```
// Almost immutable: there are holes that would
// allow state changes.
public struct PhoneList
{
    private readonly Phone[] phones;
    public PhoneList(Phone[] ph)
    {
        phones = ph;
    }
    public IEnumerable<Phone> Phones
    {
        get
        {
            return phones;
        }
    }
}
Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList(phones);
// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber();
```

数组类是引用类型。PhoneList 结构内部引用的是在对象外分配的同一存储（ Phone ）数组。开发者可以通过引用同一存储的另外引用修改这个不可变结构。为了排除这种可能，你需要被动复杂数组。前面例子就暴露了可变集合的缺陷。甚至更多糟糕的可能性存在，Phone 类是可变的引用类型。使用者可以修改集合的值，即使集合是 protected 防止任何修改。任何不可变类包含的可变引用类型都需要在构造函数中被动地复制：

```
// Immutable: A copy is made at construction.
public struct PhoneList2
{
    private readonly Phone[] phones;
    public PhoneList2(Phone[] ph)
    {
        phones = new Phone[ph.Length];
        // Copies values because Phone is a value type.
        ph.CopyTo(phones, 0);
    }
    public IEnumerable<Phone> Phones
    {
        get
        {
            return phones;
        }
    }
}
Phone[] phones2 = new Phone[10];
// initialize phones
PhoneList p2 = new PhoneList(phones);
// Modify the phone list:
// Does not modify the copy in pl.
phones2[5] = Phone.GeneratePhoneNumber();
```

当你返回可变引用类型同样要遵循这个规则。如果你在 PhoneList 结构体中增加一个属性检索整个数组，这个访问器仍然需要被动地复制。更多细节请查看原则27。

类的复杂性决定你使用三种中的哪一种初始化不可变类。 Address 结构体定义一个构造器允许使用者初始化地址。定义合理的构造函数通常是最简单的方法。

你还可以使用工厂方法初始化这个结构体。工厂使得很容易创建常用的值。.NET 框架 Color 类就是按照这个策略初始化系统颜色。静态方法 Color.FromKnowColor() 和 Color.FromName() 返回当前系统给的颜色值的复制。

第三， 对于那些需要多不操作构造不可变类的对象，你可以创建一个可变辅助类。.NET string 类遵从这个策略就有辅助类 System.Text.StringBuilder 。你使用 StringBuilder 进行多个操作创建 string 。在所有必须操作都执行之后就构建了一个 string 对象，你从 StringBuilder 获得这个不可变字符串。

不可变类是更简单，更容易维护的。不要盲目地为你的每个属性都创建 get 和 set 访问器。你的第一选择是存储数据需要不可变，原子的值类型。你可以轻易从这些实体构建更复杂的结构体。

小结：

本节对实现不可变原子值类型给了很好的方案，当希望数据是不可变或者保持原子性的，就可以派上用场了。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2079804

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

第三章 用 C# 表达设计

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

初学者用外语沟通。他们掌握单词，就可以组合起来表达他们的观点。随着初学者向专家转变，它们开始使用合适的成语和习语。语言变得更不像是外语，与人说话的时候就更高效、更清楚的。编程语言也没有什么不同的。你选择的技术和维护者，扩展者，或者是使用你的软件的开发者交流设计。C# 的类型总是存在 .NET 环境中。这个环境对所有类型做了一些假设。你违反了这些假设，你就增加你的函数不正确的可能性。

这章的原则不是软件设计——这卷都有关软件设计的刚要。而是，这些原则突出不同 C# 语言的特征如何更好的表达你设计的意图。C# 语言设计者增加语言特征使得表达现代设计习惯更加清晰。这些特征和其他语言的区别是很微妙的，而且你经常有很多选择。首先有多个选择就最好不过了，当你发现了后面介绍的区别你就应该改进你的程序。确保理解这些原则，并应用于那些可以以改进的系统。

一些语法引入了新的词汇描述你每天使用的习语。属性，索引器，事件和委托就是例子，并且类和接口是不同的：类定义类型，接口声明行为。基类声明类型并定义相关类型的公有行为。其他一些设计习惯随着垃圾回收器而改变。并且其他的习惯也随着引用类型变量而改变。

这章的建议会帮助你为你的设计选择最自然的表达。这会使得你创建的软件易于维护，易于扩展，并且易于使用。

小结：

第三章结合实际问题对软件设计给了很多建议，总之，更懂 C#，你的软件就会更易于维护，更易于扩展，更易于使用，一句哈：no zuo no die！

附上第三章目录：

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2086981

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则21：限制你的类型的可见性

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

不是每个人都需要知道每件事。你创建的类型也不是都需要 public 。只要能达到你的目的，你就应该让你的每个类都是最小的可见性。那总是会比你认为的更小的可见性。 internal 和 private 类可以实现 public 接口。所有客户可以访问 private 类的 public 函数接口。

创建 public 类型太简单不过了。而且，它经常只是一个权宜之计。很多单独的类型你都创建为 internal 。你也通过在类里面创建 protected 或者 private 类更进一步限制可见性。当你更新时，可见性越小，整个系统改变就越小。当你修改代码时，访问的代码越少，改变的地方就越少。

只需要暴露才暴露。实现 public 接口尽量访问更少的类。你会发现 Enumerator 模式的使用贯穿 .NET 框架库。System.Collections.Generic.List<T> 包含一个 private 类 Enumerator<T> ，它实现 IEnumerator<T> 接口：

```
// For illustration, not complete source
public class List<T> : IEnumerable<T>
{
    private class Enumerator<T> : IEnumerator<T>
    {
        // Contains specific implementation of
        // MoveNext(), Reset(), and Current.
        public Enumerator(List<T> storage)
        {
            // elided
        }
    }
    public IEnumerator<T> GetEnumerator()
    {
        return new Enumerator<T>(this);
    }
    // other List members.
}
```

当你写客户端代码时，不需要知道 Enumerator<T> 类。你只需要知道调用 List<T> 对象是实现 IEnumerator<T> 接口的可以调用 GetEnumerator 函数。具体哪的类型就是实现上的细节。 .NET 框架设计者对其他集合类也遵循同样的模式： Dictionary<T> 包含一个 private DictionaryEnumerator<T> ， Queue<T> 包含一个 QueueEnumerator<T> 等等。这个迭代类是 private 会有很多好处。首先，List<T> 可以代替任何实现 IEnumerator<T> 的类，并且一点都不会不明智。没有破坏任何东西。而且 Enumerator 类不需要和 公共语言规范（CLS）的保持一致。 public 接口才需要。你使用 Enumerator 类不需要知道实现类的详细细节。

创建 internal 类是一个经常被忽略限制类作用域的方法。默认情况下，大多数程序员总是创建 public 类，而没有想过其他替代方案。那是 VS.NET 向导的事情。不要不假思索地接受默认设置，你应该仔细考虑将要使用的新的类型。它是不是对客户很有用的，或是它主要用在这个程序集的内部使用？

通过接口暴露你的功能，使得你更容易创建内部类而不限制其在程序集外部的使用（查看原则26）。这类型应该是公共的，还是一个聚集的接口的一个更好的方式来描述它的功能？interal 类允许你使用不同的版本的类，只要它实现了相同的接口。来看下面的例子，考虑一类验证电话

号码的：

```
public class PhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
        // perform validation.
        // Check for valid area code, exchange.
        return true;
    }
}
```

几个月过去了，这个类还能很好的工作。后来你接到需要处理国际电话号码的需求。上面的号码验证就会失败。它只能处理 U.S. 的电话号码。你仍需要验证 U.S. 电话，但是你需要在同一个软件中使用国际版本。你最好能降低不同项目之间的耦合，而不是在同一个类增加额外的函数。你可以定义一个验证号码的接口：

```
public interface IPhoneValidator
{
    bool ValidateNumber(PhoneNumber ph);
}
```

下一步，让已经存在的号码验证器类实现这个接口，并且声明为 internal 类：

```
internal class USPhoneValidator : IPhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
        // perform validation.
        // Check for valid area code, exchange.
        return true;
    }
}
```

最后，你还可以创建国际号码验证器类：

```
internal class InternationalPhoneValidator : IPhoneValidator
{
    public bool ValidateNumber(PhoneNumber ph)
    {
        // perform validation.
        // Check international code.
        // Check specific phone number rules.
        return true;
    }
}
```

为了完成这个功能，你需要创建恰当的基于号码类型的类。你可以使用工厂模式达到这个目的。在程序集外，只有接口是可见的。这个类，具体于世界不同区域，只有在程序集内部可见。你可以为不同区域添加不同的验证类而不干扰系统里任何其他程序集。通过限制类的作用域，你就已经很好地限制因更新的代码和扩展整个系统的改变。

你也可以创建一个号码验证器的 public abstract 基类，包含了通用算法的实现。使用者可以通过可访问的基类访问 public 的功能。在这个例子中，我更喜欢使用 public 接口实现因为只有很少，共享的功能。其他情况使用 public abstract 基类会更好。无论你实现哪种方式，越少类是 public 可访问的。

此外，越少 public 类就越少的 public 区域这将有助于覆盖率单元测试。如果有更少的 public 类，就更少需要创建测试用例的 public 可访问的方法。同时，如果通过接口暴露更多的公共 API，你有自动创建一个系统，可以替代那些用于单元测试使用的备份。

你暴露给外部的那些类和接口是你的约定：你必须依靠它们。接口越混乱，你将来受到的约束就越多。暴露越少 public 类，你将来会有更多扩展和修改任何实现的选择。

小结：

这个原则很简单却很重要，在软件系统架构中相当重要，永远记住暴露越少的 public 接口，后续你的痛苦就会越少。接口定义规范，基类实现公有功能！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2083197

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则22：选择定义并实现接口，而不是基类

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

抽象基类提供的是类继承结构的公共祖先。接口描述实现类的原子级功能。两者都更有千秋，却不尽相同。接口是一种合约式的设计：实现接口的类必须提供所有期望函数的实现。抽象基类提供一组相关类的共有抽象。这是老套的，它是这样的：继承是“ is a ”的关系，接口是“ behavies like ”的关系。这些陈词滥调已经说了很久了，因为它们的结构说明了彼此的不同：基类描述的是对象是什么，接口描述的是对象的表现方式。

接口描述的是一个功能集，更像是一个合约。你可以在接口里创建任何占位符：方法，属性，索引器和事件。实现接口的类必须提供接口定义的所有元素的具体实现。你必须实现所有的方法，提供所有属性的访问器和索引器和定义所有事件。你确定并提前相同的行为到接口中。你可以使用接口作为参数和返回值。你还可以有更多机会重用代码因为不相关类可以实现同一个接口。更重要的是，其他开发者实现接口比继承基类会更容易。

你不能在接口做的是不能提供任何成员变量。接口没有任何实现，并且它们不能包含任何具体的数据成员。你的类要么全部实现接口的定义的所有元素，要么就没有实现接口。当然，你可以通过创建扩展方法让人觉得接口实现的错觉。System.Linq.Enumerable 类包含对于30个声明在 IEnumerable<T> 的扩展方法。扩展方法是实现 IEnumerable<T> 类的一剂良药。你可以查看原则8：

```
public static class Extensions
{
    public static void ForAll<T>(
        this IEnumerable<T> sequence, Action<T> action)
    {
        foreach (T item in sequence)
        action(item);
    }
}
// usage
foo.ForAll((n) => Console.WriteLine(n.ToString()));
```

抽象基类可以提供给子类一些实现，可以描述一些共有的行为。你可以指定数据成员，具体方法，实现虚函数，属性，事件和索引器。基类可以可以通过方法实现共有可重用功能。任何元素都可以是 virtual ， abstract 或 nonvirtual 。抽象基类可以通过具体行为，而接口不行。

实现重用功能还有一个好处：如果你在基类添加方法，所有的子类都自动隐式的加强。在这个意义上，随着时间的推移，，基类提供了一种高效的方式来扩展几个类的行为：基类增加和实现功能，子类就立即合并这些行为。在接口中添加添加元素会破坏所有实现这个接口的

类。它们没有包含这个元素的实现就不能通过编译。每个实现类都要更新以包含这个新元素。

如何选择抽象基类和接口其实就是一个随着时间的推移如何更好的支持你的抽象的功能的问题。接口是固定的，你发布的接口就是功能集的一个所有实现类要遵循的合约。基类可以随着时间推移而扩展。这些扩展会变成每个子类的一部分。

这两个模型可以混合复用的实现代码同时支持多个接口。.NET 框架很明显的例子就是 IEnumerable<T> 接口和 System.Linq.Enumerable 类。System.Linq.Enumerable 类包含大量定义在 System.Collection.Generic.IEnumerable<T> 接口中的扩展方法。这个分离有很重要的意义。任何类实现 IEnumerable<T> 直接就包含这些扩展方法。并且，还有额外一些没有在 IEnumerable<T> 定义的方法。这意味着开发者没有必要自己去实现这些方法。

检查实现 IEnumerable<T> 的天气观察类。

```
public enum Direction
{
    North,
    NorthEast,
    East,
    SouthEast,
    South,
    SouthWest,
    West,
    NorthWest
}

public class WeatherData
{
    public double Temperature { get; set; }
    public int WindSpeed { get; set; }
    public Direction WindDirection { get; set; }
    public override string ToString()
    {
    return string.Format("Temperature = {0}, Wind is {1} mph from the {2}",Temperature, W
    }
}

public class WeatherDataStream : IEnumerable<WeatherData>
{
    private Random generator = new Random();
    public WeatherDataStream(string location)
    {
        // elided
    }
    private IEnumerator<WeatherData> getElements()
    {
        // Real implementation would read from
        // a weather station.
        for (int i = 0; i < 100; i++)
        yield return new WeatherData
        {
            Temperature = generator.NextDouble() * 90,
            WindSpeed = generator.Next(70),
            WindDirection = (Direction)generator.Next(7)
        };
    }
    #region IEnumerable<WeatherData> Members
    public IEnumerator<WeatherData> GetEnumerator()
    {
        return getElements();
    }
    #endregion
    #region IEnumerable Members
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return getElements();
    }
    #endregion
}
```

WeatherStream 类要模拟出一系列的天气观察。为了实现这点它实现了 IEnumerable<WetherData> 。这就得实现两个方法：GetEnumerator<T> 方法和类的 GetEnumerator 方法。后者的显示实现可以让客户端代码自然把泛型对象向上转换为 System.Object 。

实现了这两个方法 WeatherStream 类支持所有在 System.Linq.Enumerable 的扩展方法。这意味着 WeatherStream 可以是 LINQ 查询的数据源：

var warmDays = from item in new WeatherDataStream("Ann Arbor") where item.Temperature > 80 select item;

LINQ 查询语法会被编译成方法调用。上面查询会被翻译为下面的调用：

var warmDays2 = new WeatherDataStream("Ann Arbor").Where(item => item.Temperature > 80). Select(item => item);

在上面的代码，Where 和 Select 的调用看起来觉得它们是 IEnumerable<WeatherData> 的方法。但其实不是。这两个方法像是属于 IEnumerable<WeatherData> 因为它们是扩展方法。它们实际是 System.Linq.Enumerable 的静态方法。编译器翻译这些调用为下面的静态调用：

```
// Don't write this, for explanatory purposes
var warmDays3 = Enumerable.Select(Enumerable.Where( new WeatherDataStream("Ann Arbor"), i
```

最后的这个版本是告诉你接口真的不可以包含实现。你通过使用扩展方法来模拟。 LINQ 就是在 System.Linq.Enumerable 类中创建 IEnumerable<T> 的扩展方法。

这让我回到使用接口左右参数和返回值的主题。接口可以任意不相关的类实现。接口的编码比基类的编码给其他开发者提供了更多的灵活性。这样非常重要所以 .NET 环境只支持单一继承关系。

下面三个方法完成的工作是相同的：

```
public static void PrintCollection<T>(IEnumerable<T> collection)
{
    foreach (T o in collection)
        Console.WriteLine("Collection contains {0}",o.ToString());
}

public static void PrintCollection(System.Collections.IEnumerable collection)
{
    foreach (object o in collection)
        Console.WriteLine("Collection contains {0}",o.ToString());
}

public static void PrintCollection(WeatherDataStream collection)
{
    foreach (object o in collection)
        Console.WriteLine("Collection contains {0}",o.ToString());
}
```

第一个方法是可重用的，任何实现 IEnumerable<T> 的类都可以使用这个方法。除了 WeahterDataStream， List<T>， SortedList<T>， 数组 和 LINQ 查询的结果都可以使用。第二个方法对很多类也有用，但是只是用在不完美的非泛型 IEnumerable 上。最后的方法是最不能重用的。它不能被数组，ArrayList， DataTable， HashTable， ImageList 和其他集合类使用。在方法编码上使用接口作为参数类是更加普遍和更加容易重用。

使用接口定义类的 API 会更具灵活性。WeatherDataStream 类可以实现返回 WeatherData 对象的集合的方法。那会是像这样的：

```
public List<WeatherData> DataSequence
{
    get { return sequence; }
}
private List<WeatherData> sequence = new List<WeatherData>();
```

这样会遗留一个很脆弱的问题。有时，你会将 List<WeatherData> 改为 数组或 SortedList<T> 。任何改变都会破坏之前的代码。当然，你可以改变参数类型，但也要改变类的 public 接口。改变类的 public 接口会引起大系统的更多改变；你需要在所有访问 public 属性的地方进行修改。

第二个问题是更直接和更令人困惑的：List<T> 类提供了改变它所包含的数据的多种方法。这个类的用户可以删除，修改甚至替换序列中的每个对象。这些绝大多数都不是你想要的。幸运的是，你可以限制使用这个类的用户的权限。为了不直接返回内部对象的直接引用，你可以返回接口给你的使用者。这其实就是返回 IEnumerable<WeatherData> 。

当你的类以类暴露属性，也就暴露了这个类的所有接口。使用接口，你就可以选择哪些方法和属性暴露给你的使用者。实现接口的类可以随着时间推移而改变实现细节。

更重要的是，不相关的类可以实现相同的接口。假设你正在构建一个应用程序管理员工，客户，和供应商。至少在类的结构是不相关的。但是它们共享了一些共有的功能。它们有名字，它们都要在程序的控制台展示名字。

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Name
    {
        get
        {
            return string.Format("{0}, {1}",LastName, FirstName);
        }
    }
    // other details elided.
}

public class Customer
{
    public string Name
    {
        get
        {
            return customerName;
        }
    }
    // other details elided
    private string customerName;
}

public class Vendor
{
    public string Name
    {
        get
        {
            return vendorName;
        }
    }
    // other details elided
    private string vendorName;
}
```

Employee， Customer 和 Vendor 类都没有继承同一个基类。但是他们分享几个属性：名字（上面展示的），地址和联系电话号码。你可以提取这些属性到接口中：

```
public interface IContactInfo
{
    string Name { get; }
    PhoneNumber PrimaryContact { get; }
    PhoneNumber Fax { get; }
    Address PrimaryAddress { get; }
}

public class Employee : IContactInfo
{
    // implementation elided.
}
```

这个接口通过让你知道构建这些不相关类的共有的任务来简化你编程的工作：

```
public void PrintMailingLabel(IContactInfo ic)
{
    // implementation deleted.
}
```

这是对于实现 IContactInfo 所有实体的例行工作。 Customer， Employee 和 Vendor 有相同的工作——但是只是因为你提取它们到接口中了。

使用接口同时衣蛾意味着你可以让结构体 省去了拆箱的操作带来的损耗。当你把结构体放入箱中，这个箱可以实现结构体支持的接口。当你通过接口指针访问结构体，你不需要进行将结构体拆箱成要访问的那个对象。例如，想象这个接口定义一个链接和一个描述：

```
public struct URLInfo : IComparable<URLInfo>, IComparable
{
    private string URL;
    private string description;
    #region IComparable<URLInfo> Members
    public int CompareTo(URLInfo other)
    {
        return URL.CompareTo(other.URL);
    }
    #endregion
    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (obj is URLInfo)
        {
            URLInfo other = (URLInfo)obj;
            return CompareTo(other);
        }
        else
            throw new ArgumentException("Compared object is not URLInfo");
    }
    #endregion
}
```

当你可以很简单地创建 URLInfo 对象的有序列表因为 URLInfo 实现了 IComparable<T> 和 IComparable 。即使代码依赖的是类 IComparable 也有更少的 封箱和拆箱次数因为使用者可以调用 IComparable.CompareTo() 而不用对对象进行拆箱操作。

基类描述和或实现相关具体子类的共有行为。接口描述的是不相关的具体类型可以实现的原子功能。都各有千秋。类定义了你创建的类是什么。接口描述实现类的功能行为。一旦你理解这些不同，你就可以创建更高效设计更好面对变化。使用类结构定义相关的类。使用接口暴露实现类的功能。

小结：

这个原则通过一系列的讲解告诉接口和类的区别，其实就是说在软件设计过程中更多关注的是对象能做什么，而不是对象是什么，越多行为的抽象，后期的问题就越少，多使用接口！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2083404

原则23：理解接口方法和虚函数的区别

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

咋一看，实现接口和重载一个虚函数似乎是一样的。都是定义一个在另一个类中声明的成员。第一眼的感觉是很有欺骗性的。实现接口和重载虚函数是非常不同的。在接口声明的成员是非虚的——至少不是默认的。子类不能重载基类实现的接口的成员。接口可以显示实现，可以把它们中 public 接口中隐藏。它们的概念不同而且使用也不同。

但是你可以这样实现接口以至于子类可以修改你的实现。你只需要对子类做一个 hook 就行了。

为了说明它们的不同，定义一个简单的几块和它的实现类：

```
interface IMsg
{
    void Message();
}
public class MyClass : IMsg
{
    public void Message()
    {
        Console.WriteLine("MyClass");
    }
}
```

Message() 方法是 MyClass 类公有接口的一部分。Message 也可以通过 IMsg 指针访问，它是 MyClass 类型的一部分。现在通过添加子类变得更复杂：

```
public class MyDerivedClass : MyClass
{
    public void Message()
    {
        Console.WriteLine("MyDerivedClass");
    }
}
```

注意到我不得不添加 new 关键字用以区别之前的 Message 方法（查看原则33）（译者注：这应该是第一版的叙述）。 MyClass.Message() 是非虚的。子类不能提供重载的 Message 版本。 MyClass 创建了新的 Message 方法，但是这个方法没有重载 MyClass.Message : 它会被隐藏。更重要的是，MyClass.Message 仍然可通过 IMsg 引用访问：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints "MyClass"
```

接口的方法是非虚的。当你实现接口，你就在这个类中声明这个合约的具体的实现。

但是你经常想要创建接口，在基类实现它们，并且在子类修改它们的行为。你确实可以做到。你有两种选择。要是你不能接触到基类，你可以在子类中重新实现接口：

```
public class MyDerivedClass : MyClass
{
    public new void Message()
    {
        Console.WriteLine("MyDerivedClass");
    }
}
```

新增的关键字使得 IMsg 改变行为子类的行为以至于 IMsge.Message 可以调用子类的版本：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints " MyDerivedClass "
```

如果你仍然使用 new 关键字在 MyDerivedClass.Message 方法上。给你个提示：仍然还会有问题（查看原则33）。子类的版本仍然可以通过子类的引用访问到：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints "MyDerivedClass"
MyClass b = d;
b.Message(); // prints "MyClass"
```

修复这个问题方法是修改基类，声明接口方法为 virtual：

```
public class MyClass : IMsg
{
    public virtual void Message()
    {
        Console.WriteLine("MyClass");
    }
}
public class MyDerivedClass : MyClass
{
    public override void Message()
    {
        Console.WriteLine("MyDerivedClass");
    }
}
```

MyDerivedClass——其他所有继承自 MyClass ——都可以声明它们自己的 Message() 方法。重载的版本总是会被调用：无论是通过 MyDerivedClass 引用，还是通过 IMsg 引用，或者是通过 MyClass 引用。

要是你不喜欢虚函数的掺杂，你只需要在定义 MyClass 上定义做一个小的变化：

```
public abstract class MyClass : IMsg
{
    public abstract void Message();
}
```

是的，你可以实现接口却没有实际实现这个接口的方法。通过声明接口方法的 abstract 版本，你就是声明继承的子类都必须实现这个接口。 IMsg 是 MyClass 声明的一部分，但是定义的方法被延迟到子类中实现。

子类可以防止进一步的重载的密封方法：

```
public class MyDerivedClass2 : MyClass
{
    public sealed override void Message()
    {
        Console.WriteLine("MyDerivedClass");
    }
}
```

另一个解决方案是实现这个的接口中调用一个虚方法，以让子类加入接口的合约中。你可以在 MyClass 中这样做：

```
public class MyClass2 : IMsg
{
    protected virtual void OnMessage()
    {
    }
    public void Message()
    {
        OnMessage();
        Console.WriteLine("MyClass");
    }
}
```

任何子类重载 OnMessage() 添加它们自己的工作到声明在 MyClass2 的 Message() 方法中。这个模式你在前面类实现 IDisposable 中见过 （查看原则17）。

显式接口实现（查看原则31）使你能够实现 一个接口，也可以隐藏你的类的公共接口。它的使用实现接口和重载虚函数变得不那么清晰。你可以使用显示接口实现限制使用者可以有访问更多的接口方法版本。 IComarable 习惯会在原则31详细展示这点。

还有最后一个添加接口和基类一起工作的惊喜。基类可以提供接口中方法的默认实现。然后，子类可以声明实现这个接口并从基类中继承这个接口的实现，正如下面例子一样。

```
public class DefaultMessageGenerator
{
    public void Message()
    {
        Console.WriteLine("This is a default message");
    }
}
public class AnotherMessageGenerator :DefaultMessageGenerator, IMsg
{
    // No explicit Message() method needed.
}
```

注意到子类可以声明接口是其的一部分合约，即使它没有提供任何 IMsg 方法的实现。只要它由恰当的公有可访问签名的方法，那么满足接口的合约。使用这个方法，你可以不用显示接口实现。

实现接口比创建和重载虚函数有更多选择。你可以创建 sealed 实现，虚实现，或者是类继承接口的抽象约束。你也可以创建 sealed 实现并提供一个虚函数调用来实现接口。你可以准确地决定怎样和什么时候子类修改你的基类实现的接口的默认行为。接口方法不是虚方法而是独立的合约。

小结：

这个原则没有大量枚举接口和继承各种组合使用的不同，那都是专牛角尖的人才去干的，而是用原理上梳理了下两者的不同，当然也有点坑需要记住的：接口的显示实现会隐藏子类的实现，添加 new 关键字可以解决这个问题，但是还不是多态。

作为接口使用，如果基类没有 virtual 和 子类也没有 new 那么基类实现优先级会更高！！！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持"分享"之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2083428

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则24：使用委托来表达回调

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

我：“儿子，到院子去除草。要去看会书。”

Scott：“爸爸，我清理好院子了。”

Scott：“爸爸，我已经把草放在除草机上了。”

Scott：“爸爸，除草机不能启动。”

我：“让我来启动它。”

Scott：“爸爸，我已经做完了。”

这个简单的交互展示了回调。我给儿子一个任务，它（重复）报告他的状态以打断我。而当我在等待他完成任务的每一个部份时，我不用阻塞我自己的进程。他可以在有重要(或者事件)状态报告时或者向我询求帮助时，可以定时的打断我，。回调就是用于异步的提供服务器与客户之间的信息反馈。它们可能在多线程中，或者可能是简单的提供一个同步更新点。在 C# 语言里是用委托来表达回调的。

委托提供类型安全的回调定义。虽然大多数是作为事件使用，那不是 C# 语言唯一使用这个特性的地方。任何时候，如果你想在两个类之间进行通信，而你又期望比使用接口有更少的偶合性，那么委托是你正确的选择。委托可以让你在运行时确定目标并且通知用户。委托就是包含了某些方法的引用的对象。这些方法可以是静态方法，也可以是实例方法。使用委托，你可以在运行时确定与一个或者多个客户对象进行交互。

回调和委托是 C# 语言常见的习惯，组合 lambda 表达式语法的以表达委托。此外，.NET 框架定义了许多常见的委托形式，如 Predicate<T>， Action<> 和 Func<> 。Predicate<T> 是测试条件的布尔函数。 Func<> 传入多个参数并产生一个单一的结果。是的，这意味着 Func<T,bool> 和 Predicate<T> 是相同的形式。尽管编译器不会将 Predicate<T> 和 Func<T,bool> 看做相同的。最后， Action<> 有多个参数并且有 void 的返回值类型。

LINQ 就是由这些概念构建出来的。 List<T> 类也包含很多使用回调的方法。看下面的代码：

```
List<int> numbers = Enumerable.Range(1, 200).ToList();
var oddNumbers = numbers.Find(n => n % 2 == 1);
var test = numbers.TrueForAll(n => n < 50);
numbers.RemoveAll(n => n % 2 == 0);
numbers.ForEach(item => Console.WriteLine(item));
```

Find() 方法传入一个委托，形式为 Predicate<int> 以检查队列中的每个元素。它是很简单的回调。 Find() 方法使用回调对每个元素进行检查，并且返回通过谓词测试的元素。编译器会将 lambda 表达式，转换为委托，并使用委托表达回调。

TrueForAll() 同样它检查每个元素，而且返回谓词为 true 的项。 RemoveAll() 移除那些谓词为 true 的项以修改队列。

最后， List.ForEach() 方法对队列的元素执行指定的动作。和前面一样，编译器转换 lambda 表示为方法并创建委托引用这个方法。

你会在 .NET 框架中发现很多这样的例子。所有 LINQ 都是构建与委托之上的。回调函数是用来处理在 WPF 和 Window Form 上的多线程编程。.NET 框架需要很简单方法的地方，它会使用委托，即调用者可以用 lambda 表达式来表达。当你需要在 API 中需要回调约定可以遵循这个例子的做法。

由于历史原因，所有委托都是多播委托。多播委托会一次调用所有添加的目标函数。有两点需要注意的：如果有异常是不安全的，而且最后执行的目标函数的返回值会作为回调的返回值。

在多播委托调用的内部，每个目标对象会被连续调用。委托不捕捉任何异常。因此，任何异常的抛出将终止委托链的调用。

返回值存在一个相似的问题。你可以定义委托有返回类型不是 void 。你可以写一个回调来检查用户中止：

```
public void LengthyOperation(Func<bool> pred)
{
    foreach (ComplicatedClass cl in container)
    {
        cl.DoLengthyOperation();
        // Check for user abort:
        if (false == pred())
            return;
    }
}
```

单一委托是可以正常工作的，但是对于多播是有问题的：

```
Func<bool> cp = () => CheckWithUser();
cp += () => CheckWithSystem();
c.LengthyOperation(cp);
```

委托调用的返回值是多播链最后一个函数调用的返回值。所有其他返回值都会被忽略。 CHeckWithUser() 位置的返回值是会被忽略的。

你可以通过自己调用目标委托解决这个问题。每个创建的委托包含一个委托队列。自己检查这个队列并遍历调用：

```
public void LengthyOperation2(Func<bool> pred)
{
    bool bContinue = true;
    foreach (ComplicatedClass cl in container)
    {
        cl.DoLengthyOperation();
        foreach (Func<bool> pr in pred.GetInvocationList())
            bContinue &= pr();
        if (!bContinue)
            return;
    }
}
```

在这个例子，我已经定义了每个委托返回值必须是 true 才会继续执行的语义。

委托提供在运行时利用回调的最好方式，满足简单的客户类上的需求。你可以在运行时确定委托目标。你可以支持多个回调目标。客户端回调需要使用 .NET 的委托实现。

小结：

使用委托来表达回调。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持"分享"之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2083564

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则25：实现通知的事件模式

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

.NET 的事件模式无非就是观察者模式的语法规范。（查看 Design Patterns, Gamma, Helm, Johnson, and Vlissides pp.293-303）事件定义类的通知消息。事件是构建在委托之上提供类型安全函数签名的处理。事实上，大多数使用委托的例子就是事件，开发者会认为事件和委托是同一件事。在原则中，我给你介绍了使用委托而不是事件的用法。当你需要通知多个客户告诉他们你的行为，你就可以触发事件。事件就是对象通知观察者。

考虑下面这个简单例子。你正要应用中构建分发消息的日志类。它可以接受所有源头的所有消息并且能够分发给监听者。这些监听者可能是控制台，或者是数据库，或者是系统日志，或者是其他机制。你如下定义这个类，当消息到来就会触发一个事件：

```
public class LoggerEventArgs : EventArgs
{
    public string Message { get; private set; }
    public int Priority { get; private set; }
    public LoggerEventArgs(int p, string m)
    {
        Priority = p;
        Message = m;
    }
}
public class Logger
{
    static Logger()
    {
        theOnly = new Logger();
    }
    private Logger()
    {
    }
    private static Logger theOnly = null;
    public static Logger Singleton
    {
        get { return theOnly; }
    }
    // Define the event:
    public event EventHandler<LoggerEventArgs> Log;
    // add a message, and log it.
    public void AddMsg(int priority, string msg)
    {
        // This idiom discussed below.
        EventHandler<LoggerEventArgs> l = Log;
        if (l != null)
            l(this, new LoggerEventArgs(priority, msg));
    }
}
```

AddMsg 方法就是恰当触发事件的方式。用临时变量引用日志事件处理器能在多线程程序中保证共享条件安全。如果没有复制引用，在 if 条件检查和事件处理执行之间可以移除事件处理器。通过复制引用，就不会发生。

LoggerEventArgs 包含事件的优先级和消息。委托定义事件处理器。在 Logger 类中，使用 event 域定义事件处理器。编译器检查到 public event 域的定义会自动为你创建 add 和 remove 操作。和你下面的写法产生的代码是一样的：

```
public class Logger
{
    private EventHandler<LoggerEventArgs> log;
    public event EventHandler<LoggerEventArgs> Log
    {
        add { log = log + value; }
        remove { log = log - value; }
    }
    public void AddMsg(int priority, string msg)
    {
        EventHandler<LoggerEventArgs> l = log;
        if (l != null)
            l(null, new LoggerEventArgs(priority, msg));
    }
}
```

C# 编译器为 event 创建 add 和 remove 访问器。我发现 public event 的声明语句会比没有 add/remove 语法更简洁，更容易阅读。所以当你声明 public event 时，让编译器为你创建 add 和 remove 属性。只有当你需要 add 和 remove 做更多事情才需要自己写。

事件不需要掌握任何潜在的监听者的信息。下面的类会自动路由所有消息到标准错误控制台：

```
class ConsoleLogger
{
    static ConsoleLogger()
    {
        Logger.Singleton.Log += (sender, msg) =>
        {
            Console.Error.WriteLine("{0}:\t{1}", msg.Priority.ToString(), msg.Message);
        };
    }
}
```

另一个类的实现是直接输出到系统事件日志中：

```
class EventLogger
{
    private static Logger logger = Logger.Singleton;
    private static string eventSource;
    private static EventLog logDest;
    static EventLogger()
    {
        logger.Log += (sender, msg) =>
        {
            if (logDest != null)
                logDest.WriteEntry(msg.Message,EventLogEntryType.Information, msg.Priorit
            };
    }
    public static string EventSource
    {
        get { return eventSource; }
        set
        {
            eventSource = value;
            if (!EventLog.SourceExists(eventSource))
                EventLog.CreateEventSource(eventSource, "ApplicationEventLogger");
            if (logDest != null)
                logDest.Dispose();
            logDest = new EventLog();
            logDest.Source = eventSource;
        }
    }
}
```

当有消息产生，事件会通知任何监听的客户端。Logger 不需要关心哪些对象有监听日志事件。

Logger 类只包含了一个事件。有些类（大多数窗口控制器）有大量的事件。在那些例子，为每个事件使用一个域是不可接受的。很多情况，在一个程序中只需要定义少量的事件。如果你遇到这种情况，你可以修改设计只需要在运行时创建的事件对象。

在核心框架的窗口控制子系统包含这样处理的例子。怎么实现呢，添加一个子系统到 Logger 类。每个子系统就是一个事件。客户端只会注册事件到它相关的子系统中。

扩展的 Logger 类有一个 System.ComponentModel.EventHandlerList， 它存储系统支持的所有事件对象。新的 AddMsg 添加 string 参数指定哪个子系统处理日志消息。如果子系统有监听者，事件就会被触发。同时，如果事件监听者对所有消息感兴趣，它的事件也会被触发：

```
public sealed class Logger
{
    private static System.ComponentModel.EventHandlerList Handlers = new EventHandlerList
    static public void AddLogger( string system, EventHandler<LoggerEventArgs> ev)
    {
        Handlers.AddHandler(system, ev);
    }
    static public void RemoveLogger(string system,EventHandler<LoggerEventArgs> ev)
    {
        Handlers.RemoveHandler(system, ev);
    }
    static public void AddMsg(string system, int priority, string msg)
    {
        if (!string.IsNullOrEmpty(system))
        {
            EventHandler<LoggerEventArgs> l = Handlers[system] as EventHandler<LoggerEven
            LoggerEventArgs args = new LoggerEventArgs( priority, msg);
            if (l != null)
                l(null, args);
            // The empty string means receive all messages:
            l = Handlers[""] as EventHandler<LoggerEventArgs>;
            if (l != null)
                l(null, args);
        }
    }
}
```

新的例子在 EventHandlerList 集合中存储独立的事件处理器。不好的是， EventHandlerList 还没有泛型版本。所以，你在这个例子会看到很多比这本书其他地方要多的类型转换。客户端代码注册具体的子系统，一个新的事件对象就产生了。相同子系统都是检索同一个事件对象。如果你类包含大量的事件接口，你就可以考虑使用事件处理器的集合。把事件对象成员的交给客户端是否注册事件处理器来决定。在 .NET 框架中，System.Windows.Forms.Control 类使用一个会隐藏所有的 event 域的更复杂变种的实现。每个事件都内部访问集合添加和移除具体的处理器。你可以查看 C# 语言规范中了解更多这个语法习惯的细节。

EventHandler 类没有更新到一个泛型版本。你不难用 Dictionary 构建一个自己的实现：

```
public sealed class Logger
{
    private static Dictionary<string,
    EventHandler<LoggerEventArgs>>
    Handlers = new Dictionary<string,
    EventHandler<LoggerEventArgs>>();
    static public void AddLogger( string system, EventHandler<LoggerEventArgs> ev)
    {
        if (Handlers.ContainsKey(system))
            Handlers[system] += ev;
        else
            Handlers.Add(system, ev);
    }
    static public void RemoveLogger(string system,EventHandler<LoggerEventArgs> ev)
    {
        // will throw exception if system
        // does not contain a handler.
        Handlers[system] -= ev;
    }
    static public void AddMsg(string system, int priority, string msg)
    {
        if (string.IsNullOrEmpty(system))
        {
            EventHandler<LoggerEventArgs> l = null;
            Handlers.TryGetValue(system, out l);
            LoggerEventArgs args = new LoggerEventArgs( priority, msg);
            if (l != null)
                l(null, args);
            // The empty string means receive all messages:
            l = Handlers[""] as EventHandler<LoggerEventArgs>;
            if (l != null)
                l(null, args);
        }
    }
}
```

泛型版本是增加代码提供事件字典和类型转换的权衡。我更喜欢泛型版本，但是它却没能够权衡。

事件提供通知监听者的标准语法。.NET 事件模式就是遵循 event 语法实现观察者模式。任意数量的客户注册处理器到事件并处理它们。这些客户不需要在编译时被知道。事件不需要关心它的订阅者就能正常工作。使用事件可以解耦通知的发送者和可能的接收者。发送者可以独立于接收者开发。事件是广播你的类发生的行为的消息的标准方式。

小结：

这个原则其实就是让大家多用 event ，可以结构消息的发送者和接收者之间关系，有点老死不相往来却彼此为了对方而活。

对于委托 delegate ， C# 衍生出了很多版本：deleage , Action , Func<> , Predicate<> 和 event 。

delegate: 就是原始的委托，可以理解为方法指针或方法的签名

Action: 是没有返回值的泛型委托

Func:有返回值的泛型委托

Predicate<>:返回值为 bool 类型的谓词泛型委托

event:对 delegate 的封装

至于，平常说的匿名函数和 Lambda 表达式跟具体函数一样都是委托的实现方式。

特别地，这个还解决我之前看别人代码的一个困苦：之前又看别人网络层代码老是会用一个临时变量缓存（上文说的用临时变量复制引用），一直不得其解。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2085830

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则26：避免返回类的内部对象的引用

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

你可能会觉得只读属性是只读的所以调用者不能修改它。不幸的是，这并不总是奏效的方法。如果你的属性返回引用类型，调用者可以访问任何 public 的对象成员，包括那些能修改属性状态。例如：

```
public class MyBusinessObject
{
    // Read Only property providing access to a
    // private data member:
    private BindingList<ImportantData> listOfData =
    new BindingList<ImportantData>();
    public BindingList<ImportantData> Data
    {
        get { return listOfData; }
    }
    // other details elided
}
// Access the collection:
BindingList<ImportantData> stuff = bizObj.Data;
// Not intended, but allowed:
stuff.Clear(); // Deletes all data.
```

MyBusinessObject 的任何使用者都可以修改你的内部数据集。你可以创建属性隐藏内部数据结构。你提供方法允许客户端只能通过这些方法操作数据，因此你可以管理内部状态的改变。只读属性打开类封装的后门。当你考虑这类问题时，你会认为它不是一个可读写的属性，而是一个只读属性。

欢迎来到一个基于引用的精彩系统。任何返回引用的成员都会返回对象的句柄。你给了调用者你的内部结构的句柄，因此调用者不再需要通过对象修改包含的引用。

显然，你需要阻止这类行为。你构建接口，并且希望使用者使用它。你不希望使用者可以在你不知情的情况下改变对象的内部状态。你有四种策略包含你的内部数据结构不被任意修改：值类型，不可变类型，接口和包装器。

值类型会被复制当客户端通过属性访问它们。客服端对复杂的类数据的任何改变，都不会影响你对象内部状态。客户端可以根据需求随意的改变复杂的数据。这不会影响你的内部状态。

不可变类型，例如 System.String 同样是安全的（查看原则20）。你返回 string ，或者其他不可变类型，很安全地知道没有客户端可以改变字符串。你的内部状态是安全的。

第三种方案是定义接口，从而允许客服端访问内部成员的部分功能（查看原则22）。当你创建一个自己的类时，你可以创建一些接口，用来支持对类的部分的功能。通过这些接口来暴露一些功能函数，你可以尽可能的减少一些对数据的无意修改。客户可以通过你提供的接口访问类的内部对象，而这个接口并不包含这个类的全部的功能。在 List<T> 中暴露 IEnumerable<T> 接口就是这个策略的例子。聪明的程序可以阻止那些猜测实现接口的对象实际类型并使用强制类型转换。但是那些那样做的程序员就是花更多时间去创建 bug，这是他们应得的。

这个在 BindingList 类会有点小麻烦会引起一些问题。因为没有泛型版本的 IBindingList，所以你需要创建两个不同的 API 方法访问数据：一个通过 IBindingList 接口支持 DataBinding，一个通过 ICollection<T> 或其他类似接口编程支持。

```
public class MyBusinessObject
{
// Read Only property providing access to a
// private data member:
    private BindingList<ImportantData> listOfData = new
    BindingList<ImportantData>();
    public IBindingList BindingData
    {
        get { return listOfData; }
    }
    public ICollection<ImportantData> CollectionOfData
    {
        get { return listOfData; }
    }
    // other details elided
}
```

在我们开始讨论如何创建一个完全只读的数据视图时以前，让我先简单的了解一下你应该如何响应客服端的修改。这是很重要的，因为你可能经常要暴露一个 IBindingList 给 UI 控件，这样用户就可以编辑数据。毫无疑问你已经使用过 Windows 表单的数据绑定，用来给用户提供对象私有数据编辑。BindingList<T> 类实现 IBindingList 接口，所以你响应展示给用户的集合的任何添加，更新，或者删除元素的操作。

任何时候，当你期望给客户端提供修改内部数据的方法时，都可以扩展这个的技术，但你要验证而且响应这些改变。你的类订阅对内部数据结构产生改变的事件。事件处理器验证改变或者响应这些改变以更新其他内部状态。

回到开头的问题上，你想让客户查看你的数据，但不许做任何的修改。当你的数据存储在一个 BindingList<T> 里时，

你可以通过强制在 BindingList 上设置一些属性（ AddEdit， AllowNew， AllowRemove 等）。这些属性的值被 UI 控件控制。UI 控件基于这些属性值开启和关闭不同的行为。这些是 public 的属性，所以你可以修改集合的行为。但是那样也还没有作为 public 属性暴露 BindingList<T> 对象。客户端可以修改你的属性并且规避使用只读绑定集合的意图。再强调一次，通过接口类型而不是类类型暴露内部存储可以限制客服端代码在这个对象上的行为。

最后一个选择是提供一个包装器对象并且值暴露这个包装器实例，这可以减少访问内部对象。 System.Collections.ObjectModel.ReadOnlyCollection<T> 类就是包装集合并暴露一个只读版本的数据的标准方法：

```
public class MyBusinessObject
{
    // Read Only property providing access to a
    // private data member:
    private BindingList<ImportantData> listOfData = new BindingList<ImportantData>();
    public IBindingList BindingData
    {
        get { return listOfData; }
    }
    public ReadOnlyCollection<ImportantData>CollectionOfData
    {
        get
        {
            return new ReadOnlyCollection<ImportantData>(listOfData);
        }
    }
    // other details elided
}
```

通过 public 接口直接暴露引用类型将允许使用者修改对象的内部而不通过你定义的方法或属性。这看起来不可思议，确实一个常见的错误。你应该考虑到你暴露的是引用而不是值，因此需要修改类的接口。如果你只是简单的返回内部数据，那么你就给了访问它们包含的常用的权限。客户端可以调用可访问的方法。你要限制访问private 内部数据要通过接口，包装器对象或值类型。

小结：

这里说的确实是引用类型系统或者很多需要统一管理模型的一个通病，怎么才能做到对引用类型内部改变“一夫当关万夫莫开”的效果，目前比较好的方法是使用接口！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2086266

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则27：总是使你的类型可序列化

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

持久化是类型的核心特征。没有人会注意到除非你没有支持它。如果你的类型没有支持恰当支持序列化，你就会给想要使用你的类作为成员或基类的开发者增加工作。当你的类没有支持序列化，它们必须围着它添加自己对这个标准特征的实现。当你的类有不能访问的 private 细节时，就不可能正确的实现序列化。因此，如果你不提供序列化机制，使用者是很难甚至是不可能实现这个机制。

相反，实践中要为你的类添加序列化。对于那些除了不展示 UI 组件，窗口或表单的所有类是很有实践意义的。感觉到有额外工作量是没理由的。.NET 序列化支持是那么简单的让你没有任何理由去支持它。大多数情况下，添加 Serializable 特性就足够了：

```
[Serializable]
public class MyType
{
    private string label;
    private int value;
}
```

添加 Serializable 特性就可以工作是因为这个类的所有成员都是可序列化的：string 和 int 都支持 .NET 序列化。这就是为什么都要支持序列化的原因，当你添加自定义类的域是就会变得更加明显：

```
[Serializable]
public class MyType
{
    private string label;
    private int value;
    private OtherClass otherThing;
}
```

这里的 Serializable 特性只有当 OtherClass 类支持 .NET 序列化才有效。如果 OtherClass 不可序列化，会报一个运行时错误，所以你就不得不自己写代码序列化 MyType 和内部的 OtherClass 对象。如果没有掌握 OtherClass 内部定义是不可能。

.NET 序列化会将对象的所有成员保存到输出流中。此外，.NET 序列化支持任意的对象关系图：即使对象有环引用，序列化和反序列化方法都只会保存和存储每个实际对象一次。当 web 对象反序列化时， .NET 的序列化框架同时也会重创建 web 对象的引用。当对象关系图被反序列化，任何 web 相关的对象都会被正确创建。最后重要一点是 Serializable 特性同时支持二进制和 SOAP 序列化。这个原则里介绍所有技术都支持这两种序列化机制。但是记住

只有对象关系图中所有类型都支持序列化机制才有用。那就是为什么在所有类型支持序列化重要的原因。一旦你遗漏一个，你就在对象关系图中留了一个孔，就使得使用你的类的其他人很难轻易支持序列化。不久以后，每个人都会发现不得不自己写实现序列化的代码。

添加 Serializable 特性是序列化对象的最简单技术。但是最简单的解决方法不总是最合适的解决方案。有时，你希望序列化对象的所有成员：有些成员可能只会存在长期操作的缓存中。其他成员可能持有只会在内存操作的运行时资源。你同样可以使用特性可以管理所有可能。添加 [NonSerialized] 特性给任意数据成员就不会作为对象状态的一部分保存。这就它们变得不可序列化：

```
[Serializable]
public class MyType
{
    private string label;
    [NonSerialized]
    private int cachedValue;
    private OtherClass otherThing;
}
```

不可序列化成员会增加你一点工作量。序列化 API 在反序列化时是不会初始化不可序列化成员。任何构造函数会被调用，成员变量的初始化也不会被执行。当你使用序列化特性，不可序列化成员会获得系统默认初始化值：0或 null 。如果默认0初始化不正确，你需要实现 IDeserializationCallBack 接口来初始化这些不可序列化成员。 IDeserializationCallBack 包含一个方法： ONDeserialization 。框架当整个对象关系图都反序列化之后会调用这个方法。 你使用这个方法初始化对象的不可序列化成员。因为整个关系图已经被读取，你调用类型上任何函数或序列化成员是安全的。不幸的是，它不是万无一失的。在整个对象关系图都读取之后，框架会调用每个实现了 IDeserializationCallBack 对象的 OnDeserialization 方法。在执行 OnDeserialization 可以调用关系图中的任何其他对象的 public 成员。如果它们先执行了，你对象的非序列化成员会是 null 或0。调用顺序是无法保证的，所以你必须保证所有 public 方法都要处理不可序列化成员没有被初始化的情况。

到目前为止，你已经知道为什么要对所有类型添加序列化：非不可序列化类型在序列化类型中使用会带来更多工作。你已经学会使用特性的最简单序列化方法，包括怎么初始化不可序列化成员。

序列化数据会在成员的不同版本存在。给你的类型加上序列化意味着将来你需要读取旧的版本。反序列化时如果发现类有域被添加或移除，就会抛出异常。当你需要支持多个版本你就需要更多控制序列化进程，实现 ISerializable 接口。这接口定义一些 hook 用于自己定义类的序列化机制。 ISerializable 接口的方法和存储和默认序列化的方法和存储是一致的。这说明当你创建类时可以继续使用序列化特性。如果你觉得需要提供自己的扩展时，你可以添加 ISerializable 接口的支持。

例如，考虑你如何来支持 MyType 的第2个版本，也就是添加了另一个域到类中时。简单的添加一个域都会产生一个新的类型，而这与先前已经存在磁盘上的版本是不兼容的：

```
[Serializable]
public class MyType
{
    private MyType(SerializationInfo info, StreamingContext cntxt)
    {
    }
    private string label;
    [NonSerialized]
    private int value;
    private OtherClass otherThing;
    // Added in version 2
    // The runtime throws Exceptions
    // with it finds this field missing in version 1.0
    // files.
    private int value2;
}
```

你可以添加 ISerializable 来解决这个行为。 ISerializable 接口定义一个方法，但是你不得不实现两个。 ISerializable 定义 GetObjectData() 方法写数据到流中。此外，你必须提供一个序列化构造器从流中 初始化对象：

```
private MyType(SerializationInfo info, StreamingContext cntxt)
```

下面的序列化构造函数演示了如何从先前的版本中读取数据，以及读取当前版本中的数据和默认添加的Serializable特性生成的序列化保持一致：

```
using global::System.Runtime.Serialization;
using global::System.Security.Permissions;
[Serializable]
public sealed class MyType : ISerializable
{
    private string label;
    [NonSerialized]
    private int value;
    private OtherClass otherThing;
    private const int DEFAULT_VALUE = 5;
    private int value2;
    // public constructors elided.
    // Private constructor used only
    // by the Serialization framework.
    private MyType(SerializationInfo info, StreamingContext cntxt)
    {
        label = info.GetString("label");
        otherThing = (OtherClass)info.GetValue("otherThing",typeof(OtherClass));
        try
        {
            value2 = info.GetInt32("value2");
        }
        catch (SerializationException)
        {
            // Found version 1\.
            value2 = DEFAULT_VALUE;
        }
    }
    [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    void ISerializable.GetObjectData(SerializationInfo inf,StreamingContext cxt)
    {
        inf.AddValue("label", label);
        inf.AddValue("otherThing", otherThing);
        inf.AddValue("value2", value2);
    }
}
```

序列化流把每项当做 key/value 对存储。默认特性生成的代码是使用变量名作为键值存储值。当你添加 ISerializable 接口，你必须匹配键名和变量的顺序。这个顺序就是类中声明它们的顺序。（顺便说下，这个实际说明类中变量声明顺序变了或者重命名了就破坏了已经存储的序列化文件的兼容性。）

同时，我已经要求 SerializationFormatter 的安全许可。如果没有恰当的保护，GetObjectData 可能是一个安全漏洞。恶意代码可以创建一个 StreamingContext， 并且使用 GetObjectData 获得对象中的值，或者续断修改 SerializationInfo 的版本，或者重写组织修改的对象。这可能运行恶意开发者访问你对象的内部状态，在流中修改，并且返回给你。要求 SerializationFormatter 安全许可进而就封闭了这个潜在的漏洞。这就确保只有被信任的代码才能访问对象的内部状态。

但是实现 ISerializable 接口有一个弊端。你可以看到，我之前使得 MyType 为 sealed 。就强制它是一个叶节点类。在基类实现 ISerializable 接口就要复杂到考虑所有子类。实现 ISerializable 意味着每个子类都必须创建一个 protected 构造方法用于反序列化。另外，为了支持非封闭类，你需要在 GetObjectData 方法中创建 hook ，让子类可以添加自己的数据到流中。编译器不会捕获出现的错误。当从流中读取子类时，如果没有恰当的构造函数会抛出运行时异常。缺少 GetObjectData() 的钩子意味着子类派生的数据不会被保存到文件中。没有错

误抛出。我很想建议地说“在叶节点实现实现可序列化”。但我没有那样说因为那不能正常工作。你的基类必须为子类实现可序列化。修改 MyType 使得它成为一个可序列化基类，莫修改序列化构造函数为 protected 并且创建一个虚方法子类可以重载自己的版本存储数据：

```
[Serializable]
public class MyType : ISerializable
{
    private string label;
    [NonSerialized]
    private int value;
    private OtherClass otherThing;
    private const int DEFAULT_VALUE = 5;
    private int value2;
    // public constructors elided.
    // Protected constructor used only by the
    // Serialization framework.
    protected MyType(SerializationInfo info, StreamingContext cntxt)
    {
        label = info.GetString("label");
        otherThing = (OtherClass)info.GetValue("otherThing",typeof(OtherClass));
        try
        {
            value2 = info.GetInt32("value2");
        }
        catch (SerializationException e)
        {
        // Found version 1\.
        value2 = DEFAULT_VALUE;
        }
    }
    [SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter = true)]
    void ISerializable.GetObjectData(SerializationInfo inf, StreamingContext cxt)
    {
        inf.AddValue("label", label);
        inf.AddValue("otherThing", otherThing);
        inf.AddValue("value2", value2);
        WriteObjectData(inf, cxt);
    }
    // Overridden in derived classes to write
    // derived class data:
    protected virtual void
    WriteObjectData(SerializationInfo inf,StreamingContext cxt)
    {
        // Should be an abstract method,
        // if MyType should be an abstract class.
    }
}
```

子类可以提供自己的序列化构造函数和重载 WriteObjectData 方法：

```
public class DerivedType : MyType
{
    private int derivedVal;
    private DerivedType(SerializationInfo info,
    StreamingContext cntxt) : base(info, cntxt)
    {
        derivedVal = info.GetInt32("_DerivedVal");
    }
    protected override void WriteObjectData(SerializationInfo inf, StreamingContext cxt)
    {
        inf.AddValue("_DerivedVal", derivedVal);
    }
}
```

从序列化流中写入和检索值的顺序必须保持一致。我首先对基类的值进行读和写因为我相信它是最简单的。如果你不按照继承关系中正确的顺序去读和写，序列化代码就失败。

本原则的例子代码中都没有使用自动（隐式）属性。这就是设计。自动属性使用编译器产生的支持域来存储。你不能访问支持域，因为域的名字是无效的 C# 记号（它是一个有效的 CLR 符号）。这就使得二进制序列化对使用自动属性的类而言非常脆弱。你不能写自己的序列化构造函数，或 GetObjectData 方法访问那些支持域。这样只能对简单类型有效，任何子类和增加域都会失败。随着时间推移，你会发现这个问题，并且你不能修复这个问题。任何时候你给你的类型添加 Serializable 特性，你必须使用自己的支持域存储具体地实现属性。

.NET 框架提供简单，标准算法序列化对象。如果你的类型需要持久化，你应该遵从标准的实现。如果你的类型没有支持序列化，使用这个类的其他类同样也不支持序列化。尽量为你的客户端支持这个机制。尽可能的使用默认序列化特性，并且在默认的特性不满足时要实现 ISerializable 接口。

小结：

虽然序列化的实现机制很简单，但是细节还是有很多讲究的。累死了，打了这么多字，跑步去，加油！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2086349

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则28：创建大粒度的网络服务 APIs

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

通信协议的花费和不便决定了你将会怎么使用这些媒介。使用电话，传真，信件和电子邮件交流是不同的。回想上次你从商品目录中下单。当你用电话下单，你和售货员进行问-答的交互：

“我能知道你第一选项么？”

“我选的序号是123-456。”

“你需要多少份？”

“3。”

这段交流直到售货员知道整个订单，你的账单地址，你的信用卡信息，你的送货地址，和其他完成交易的信息才终止。在电话上来回讨论是非常方便的。你不会一直自言自语而没有反馈。你不能忍受长时间的沉默如果销售人员仍然还在。

用传真订单会不一样。你会填满整个文档然后在把整个文档传真给公司。一个文档，一次交易。你不会填一栏，传真一次，填写地址，又传真一次，添加信用卡新，又传真一次。

这说明了缺乏定义的服务接口常见的误区。不管你使用 web 服务，.NET 远程，或基于 Azure 编程，你必须记住在两台机器传递对象的操作的昂贵的花费。你必须停止创建的远程 API 只是对本地接口的一个封装。这就像是用打电话处理本要用传真处理的订单。你的程序每次都要等待在网络管道传递信息往返的时间。更大粒度的 API，程序的更高比例时间花在等待数据从服务器返回。

相反，在客户端和服务器之间创建基于 web 的接口要基于一系列文档和对象集。你的远程交流应该向像用传真向商品公司下订单一样工作：客服端机器应该能在没有和服务器交流的时间工作。然后，等所有交易信息填写完毕，客服端发送整个文档给服务器。服务器用同样的方式工作：从服务器发送信息到客服端，客服端接受所有必要信息并接着完成所有任务。

接着消费者下单的比喻。我们设计客户下单处理系统，它由中心服务器和桌面客服端，彼此通过 web 服务访问信息。系统中的一个类就是 Customer 类。如果你忽略了运输问题，客户类可能是这样子的，它允许客户端代码检索或修改的名称，送货地址，和帐户信息：

```
public class Customer
{
    public Customer()
    {
    }
    // Properties to access and modify customer fields:
    public string Name { get; set; }
    public Address ShippingAddr { get; set; }
    public Account CreditCardInfo { get; set; }
}
```

Customer 类没有包含会被远程调用的 API 。调用远程的 Customer 会导致在客服端和服务器之间过度的交流：

```
// create customer on the server.
Customer c = Server.NewCustomer();
// round trip to set the name.
c.Name = dlg.Name;
// round trip to set the addr.
c.ShippingAddr = dlg.ShippingAddr;
// round trip to set the cc card.
c.CreditCardInfo = dlg.CreditCardInfo;
```

取而代之，我们会创建一个局部 Customer 对象，并且当这个对象所有域都设置好之后传递给服务器：

```
// create customer on the client.
Customer c2 = new Customer();
// Set local copy
c2.Name = dlg.Name;
// set the local addr.
c2.ShippingAddr = dlg.ShippingAddr;
// set the local cc card.
c2.CreditCardInfo = dlg.CreditCardInfo;
// send the finished object to the server. (one trip)
Server.AddCustomer(c2);
```

这个客户的例子说明一个明显又简单的例子：在客服端和服务器之间来回传递整个对象。但是为了编写高效的程序，你需要扩展这个简单例子以囊括相关的对象集。远程调用一个对象的单个属性粒度太小了。但是一个客户也可能不是在服务器和客服端交易合适的粒度。

扩展这个例子到实际的设计问题是你会在程序中遇到的，我们对系统进行些假设。这个软件系统支持主要在线厂商和一百万客户的交易。想象下大多数客户都在家里下单，去年平均15单。每个电话操作员每次都要同一台机器上下切换记录客户回答的信息。你的设计任务是决定更多高效对象集在客服端机器和服务器之间传递。

你开始就可以消除一些明显的选择。检索每个客户和每个订单是很明确被禁止的：一百万的客户和一千五百万的订单记录数据大到不可能传递给每个客户端。你需要为一个瓶颈去权衡其他。为了不每个可能的数据更新不断轰击你的服务器，你向服务器发送一个超过一千五百万的对象请求。当然，这只是一个交易，但这是一个非常低效的交易。

相反，考虑如何最好的检索一组对象集，操作者接下来的几分钟必须使用一个近似的对象的数据集。操作者会接电话并和客户交流。在通话过程中，操作者可能添加和移除订单，改变订单，或者修改客户的账户信息。显而易见的选择是一次检索整个客户的所有订单。服务器的方法是下面这样的：

```
public OrderDataCollection FindOrders(string customerName)
{
    // Search for the customer by name.
    // Find all orders by that customer.
}
```

这样就正确了？那些已经发货或已经被签收的订单大多数情况下客户端几乎不需要。一个请求客户更好的检索只是开放的订单。服务器的方法会改成这样：

```
public OrderData FindOpenOrders(string customerName)
{
    // Search for the customer by name.
    // Find all orders by that customer.
    // Filter out those that have already
    // been received.
}
```

你仍然在每个客户电话的开始就请求数据。有没有方法优化下载客户端订单信息的通信。我们继续添加一些业务流程的假设，你会得到一些想法。假设呼叫中心划分使每个工作团队只会接到一个区域代码。现在你可以修改你的设计优化传播得更多。

每个操作者会在开始切换的区域代码而检索被更新的客户和订单信息。每次电话后，客户端会将修改数据推送会服务器，服务器会对后面的客户端请求推送改变的数据。这样的结果是每次电话后，操作者会发送修改的数据给服务器并从服务器获得同组的其他操作者修改的数据。这个设计意味着每个电话只能进行一个交易，每个操作者在回答电话时总是获得正确的数据集。这就每个电话只有一个来回。现在服务器包含下面两个方法：

```
public CustomerSet RetrieveCustomerData(AreaCode theAreaCode)
{
    // Find all customers for a given area code.
    // Foreach customer in that area code:
    // Find all orders by that customer.
    // Filter out those that have already
    // been received.
    // Return the result.
}
public CustomerSet UpdateCustomer(CustomerData updates, DateTime lastUpdate, AreaCode the
{
    // First, save any updates.
    // Next, get the updates:
    // Find all customers for a given area code.
    // Foreach customer in that area code:
    // Find all orders by that customer that have been
    // updated since the last time. Add those to the result.
    // Return the result.
}
```

但是你可能还会浪费一些带宽。当每个已知客户每个都打电话来下单时，你最后的设计作品效果最好。这是不正确的。如果是，你的公司已经远远超出一个软件项目范围的客户服务问题。

在不增加事务请求数量或服务响应客户的延迟的情况下，我们怎样才能进一步限制每笔交易的数据的大小？你可以对数据库中的客户进一步假设。你跟踪的一些统计并发现如果客户六个月不订购，他们不可能再次订购。所以从那日期你可以停止这些客户和他们的订单。这会缩小初始化交易的大小。你还发现很多客户在下了单之后在打电话来通常是询问上一次订单。所以你修改发送给客户端的订单信息只是上一次的而不是所有的订单列表。者不需要改服务器方法的前面，但是会减少发给客户端数据包的大小。

这一假设的讨论集中在让你们去思考远程机器之间的通信：你要最小化机器之间通信的频率与传输的大小。这两个目标是相违背的，你需要在它们之间做出权衡。你放弃以两个极端的中心的做法，但是大通信量会有相对更少的负面影响。

小结：

这个原则介绍的如果服务器和客户端高效的通信，数据量和频率——其实就是一次数据请求的完整性和网络带宽的权衡，这么大的问题，放在一个原则来讲，有点华而不实。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2086414

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则29：让接口支持协变和逆变

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

类型可变性，具体地，协变和逆变，定义了一个类型变化为另一个类型的两种情况。如果可能，你应该让泛型接口和委托支持泛型的协变和逆变。这样做可以让你的 APIs 能安全地不同方式使用。如果你不能将一个类型替换为另一个，那么就是不可变。

类型可变性是很多开发者遇到的却又不真正理解的很多问题之一。协变和逆变是类型替换的两种不同形式。如果你用声明类型的子类返回那么就是协变的。如果你用声明类型的基类作为参数传入那么就是逆变。面向对象原因普遍支持参数类型的协变。你可以传递子类对象到任何期望是基类参数的方法。例如， Console.WriteLine() 函数有一个使用 System.Object 参数的版本。你可以传入任何 System.Object 的子类对象。如果你重载实例方法返回 System.Object， 你可以返回任何继承自 System.Object 的对象。

普遍的行为让很多开发者认为泛型也遵循这个规则。你可以使用 IEnumerable<MyDerived> 传给参数为 IEnumerable<Object> 的方法。你会期望返回的 IEnumerable<MyDerivedType> 可以赋值给 IEnumerable<Object> 变量。不是这样的。在 C# 4.0之前，所有泛型类型都是不可变的。这意味着，很多次你都自以为泛型也有协变和逆变时，编译器却告诉你的代码是有问题的。数组是被看做协变的。然而，数组不支持安全的协变。随着 C# 4.0， 新关键字可以让你的泛型支持协变和逆变。这使得泛型更有用，特别是在泛型接口和委托上你应该尽可能使用 in 和 out 参数。

我们开始通过数组理解协变的问题。考虑下面简单的类继承结构：

```
abstract public class CelestialBody
{
    public double Mass { get; set; }
    public string Name { get; set; }
    // elided
}
public class Planet : CelestialBody
{
    // elided
}
public class Moon : CelestialBody
{
    // elided
}
public class Asteroid : CelestialBody
{
    // elided
}
```

下面这个方法把 CelestialBody 对象数组当做协变，而且那样做事安全的：

```
public static void CoVariantArray(CelestialBody[] baseItems)
{
    foreach (var thing in baseItems)
        Console.WriteLine("{0} has a mass of {1} Kg", thing.Name, thing.Mass);
}
```

下面这个方法也把 CelestialBody 对象数组当做协变，但这是不安排的。赋值语句会抛出异常。

```
public static void UnsafeVariantArray(
CelestialBody[] baseItems)
{
    baseItems[0] = new Asteroid
    { Name = "Hygiea", Mass = 8.85e19 };
}
```

如果你将子类赋给基类的数组元素一样会有相同的问题：

```
CelestialBody[] spaceJunk = new Asteroid[5];
spaceJunk[0] = new Planet();
```

把集合看着协变意味着当如果有两个类有继承关系是，你可以认为他们的关系和两个类型的数组是一样的。这不是一个严格的定义，但要记住它是很用的。 Planet 对可以传递给任何期望参数为 CelestialBody 的方法。这是因为 Planet 继承于 CelestialBody 。类似地，你可以将 Planet[] 传递给任何期望参数为 CelestianlBody[] 的方法。但是，正如上面的例子一样，它们总是不能如你期望一样工作。

当泛型被引入时，这个问题被十分严格的处理。泛型总是被当做不可变的。泛型类型不得不正确匹配。然而，在 C# 4.0，你可以将方向接口修饰变为协变或逆变。我们先讨论泛型协变，而后在讨论逆变。

下面这个方法调用参数为 List<Planet> ：

```
public static void CoVariantGeneric(
IEnumerable<CelestialBody> baseItems)
{
    foreach (var thing in baseItems)
        Console.WriteLine("{0} has a mass of {1} Kg", thing.Name, thing.Mass);
}
```

这是因为 IEnumerable<T> 已经被扩展为限制 T 只能出现在输出位置：

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> :IDisposable, IEnumerator
{
    T Current { get; }
        // MoveNext(), Reset() inherited from IEnumerator
}
```

我给出了 IEnumerable<T> 和 IEnumerator<T> 的定义，因为 IEnumerator<T> 会有比较重要的限制。注意到 IEnumerator<T> 现在的参数类型 T 已经被修饰符 out 修饰。这就强制编译器类型 T 只能在输出位置。输出位置仅限于函数返回值，属性 get 访问器和委托的参数。

因此，使用 IEnumerable<out T>，编译器知道你会查看序列的每个 T，但是不会修改序列的内容。这个例子中把 Planet 当做 CelestailBody 就是这样的。

IEnumerable<T> 可以协变是因为 IEnumerator<T> 也是协变的。如果 IEnumerable<T> 返回的接口不是协变的，编译器会产生一个错误。协变类型必须返回值类型的参数或这个接口是协变的。

然而，下面方法替换队列的第一个元素的泛型是不可变的：

```
public static void InvariantGeneric(
IList<CelestialBody> baseItems)
{
    baseItems[0] = new Asteroid { Name = "Hygiea", Mass = 8.85e19 };
}
```

因为 IList<T> 的参数 T 既没有被 in 又没有被 out 修饰符，你必须使用正确的类型进行匹配。

当然，你也可以创建逆变泛型接口和委托。用 in 修饰符替换 out 。这个告诉编译器类型参数只能出现在输入位置。.NET 框架已经为 IComparable<T> 加上了 in 修饰符：

```
public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

这说明如果 CelestialBody 实现 IComparable<T>，可以使用很多不同的对象。它可以比较两个 Planet，一个 Planet 和一个 Moon，一个 Moon 和一个 Asteroid，或者其他组合。比较了多个不同的对象，但这是有效的比较。

你会注意到 IEquatable<T> 是不可变的。按照定义， Planet 对象不会和 Moon 对象相等。它们是不同的类型，所以没有意义。如果两个对象是相同类型的如果相等而且不充分的，它是必要的（查看原则6）。

类型参数是可逆变的只有作为方法参数或某些地方的委托参数。

现在，你应该已经注意到我已经用了词组“某些地方的委托参数”两次。委托的定义可以协变也可以逆变。这是相当简单：方法参数逆变（ in ），方法的返回值是协变（ out ）。BCL 更新了包括下面变种的很多委托的定义：

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, T2, out TResult>(T1 arg1,T2 arg2);
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, T3>(T1 arg1, T2 arg2, T3 arg3);
```

在重复一次，这也许不太难。但是，如果你把它们混淆了，事情就得开动你的脑筋了。你已经看到你不能从协变接口返回不可变接口。你使用委托要么限制协变要么限制逆变。

如果你不仔细的话，委托在接口里会向协变和逆变偏移。这里有几个例子：

```
public interface ICovariantDelegates<out T>
{
    T GetAnItem();
    Func<T> GetAnItemLater();
    void GiveAnItemLater(Action<T> whatToDo);
}
public interface IContravariantDelegate<in T>
{
    void ActOnAnItem(T item);
    void GetAnItemLater(Func<T> item);
    Action<T> ActOnAnItemLater();
}
```

接口里的方法的命名展示了它们具体的工作。仔细看 ICovariantDelegate 接口的定义。GetAnItemLater() 只是检索元素。方法中可以调用 Func<T> 返回检索的元素。 T 仍然出现在输出位置上。这可能是有意义。 GetAnItemLater() 很容易让人困扰。这里，你的委托方法只是接收 T 对象。所以，即使 Action<T> 是协议的，它出现的 ICovarinatDelegate 接口的位置其实是 T 由实现 ICovariantDelegate<T> 的对象返回的。它看起来是逆变的，但是相对于接口来说是协变的。

IContravariantDelegate<T> 和一般的接口一样但是展示如何使用逆变接口。再说一次，ActOnAnItemLater() 方法就很明显。 ActOnAnItemLater() 方法有些复杂。你返回一个接受 T 类型对象的方法。这个最后方法，一次又一次强调，会引起一些困扰。它和其他接口的概念是一样的。 GetAnItemLater() 方法接受一个方法并返回 T 对象。即使 Func<out T> 声明为协变，它的作用是为实现 IContravariantDelegate 对象引入输入。它相对于 IContravariantDelegate 的作用是逆变的。

描述协变和逆变如何正确的工作十分复杂。值得庆幸的是，语法现在支持使用 in （逆变） 和 out （协变）修饰接口。你应该尽可能使用 in 或 out 修饰符修复接口和委托。然后，编译器就会纠正和你定义的有差异的用法。编译器会捕获到接口和委托的定义，并且发现你创建的类型的任何误用。

小结：

这个原则作为第三章的最后一个，虽然介绍的是类型的可变性，有些类似类型转换，但是情况却复杂的多，理解起来难度很大，想要更彻底的理解协变和逆变的概念，可以参考①。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

参考：

①1-2-3.cnblogs.com： http://www.cnblogs.com/1-2-3/archive/2010/09/27/covariance-contravariance-csharp4.html

第四章 和框架一起工作

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

在2002年 .NET 第一个发布，我的朋友兼同事 Martin Shoemaker 组织过一次圆桌会议研究讨论“我必须写这样的 .NET 代码吗？”在那时这是一个伟大的圆桌会议，现在也更相关了。.NET 框架已经进步了，而且比那时包含更多的新类和特性。重要的是你不用再创建已经存在的特性。

.NET 框架是一个丰富的类库。你对框架了解的越多，你自己需要写的代码就越少。这个框架类库为你做了更多工作。不好的是，基础类库需要应付版本4的相关问题。有更好的方式解决在版本1遗留的问题。但是框架团队不会删除这些就的 API 和类。甚至都没有将那些旧的 API 标记为过时的。它们仍然可以工作，而且你也不会对重写正在工作的代码感兴趣。但是当你要创建信的嗲吗，你应该使用已存在的最好的工具。这章展示给你 .NET 框架版本4.0的技术。当你面临框架多个选择可用时，有一些原则可以帮助你做最好的选择。还有一些原则解释一些技术，你可以使用它们，当你想要你的类更好和框架设计者创建的类协作。

小结：

第四章主要是如何充分利用 C# 4.0框架实现的功能，减少自己的实现。正如上面说的，你对 .NET 了解的越多，你需要实现的特性就越少。很多模式和机制，.NET 都已经提供了类和 API 实现了，不需要操心太多，我们只需要理解它们的原理，才会用的更好！因为工作中还没有涉及到对 PLINQ 的使用，但是在翻译的过程中还是受益颇多。

附上第四章目录：

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

原则30：选择重载而不是事件处理器

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

很多 .NET 类提供两种方式处理系统事件。你可以附加指定一个事件处理器或者重载基类的虚函数。为什么提供两种方式做同一件事情？因为不同的情况需要不同的解决方案，这就是为什么。在子类中，你总可以重载虚函数。这样不相关对象就不能使用这个事件处理器。你可以写的一个漂亮的 Windows Presentation Foundation (WPF) 应用，需要响应鼠标按下事件。在你的 Form 类中，你可以选择重载 OnMouseDown 方法：

```
public partial class Window1 : Window
{
    // other code elided
    public Window1()
    {
        InitializeComponent();
    }
    protected override void OnMouseDown(MouseButtonEventArgs e)
    {
        DoMouseThings(e);
        base.OnMouseDown(e);
    }
}
```

或者，你也可以指定一个事件处理器（需要 C# 和 XAML ）：

```
<!-- XAML File -->
<Window x:Class="Item36_OverridesAndEvent.Window1" xmlns= "http://schemas.microsoft.com/w
    <Grid>
    </Grid>
</Window>

// C Sharp file:
public partial class Window1 : Window
{
    // other code elided
    public Window1()
    {
        InitializeComponent();
    }
    private void OnMouseDown(object sender,MouseButtonEventArgs e)
    {
        DoMouseThings(e);
    }
    private void DoMouseThings(MouseButtonEventArgs e)
    {
        throw new NotImplementedException();
    }
}
```

第一个方案是更好的。WPF 程序中声明代码是重点这得可能令人惊讶。即使这样，如果逻辑代码需要用代码实现，你应该使用虚函数。如果一个事件处理器抛出异常了，在链中的其他处理器就不会被调用（查看原则24和25）。通过重载 protected 需函数，你的处理器就可以被先调用。基类的虚函数版本是负责处理协议特殊的事情。这意味着如果你想要事件处理器被调用（并且你总是这样做的），你必须调用基类的虚函数。在一些少数的情况，你不想要默认的行为，你就不调用基类的版本这样就没有处理器被调用。你不能保证所有的事件处理器会被调用因为有些不正确的事件处理器会抛出异常，但是你可以保证你的子类行为是正确的。

使用重载比附加事件处理器更高效。你应该记得原则25讲到的事件是一个多播委托。这就使得一个事件源会有多个观察者。这个事件机制会占用处理器多点时间，因为他必须检查事件是否有添加事件处理器。如果有，它必须遍历整个调用队列，而这个队列可能包含了任意数量的目标函数。队列的每个方法都会被调用。检查是否有事件处理器并在运行时遍历调用会比只调用一个虚函数话费更多时间。

如果这个还不够，再查看下上面的两个例子。哪个更清晰？重载一个虚函数只需要检查一个函数并且如果维护这个表单只需要修改这个函数。事件机制有两个地方需要维护：事件处理器函数和关联事件的代码。这两个都会导致事件失败。一个函数就很更简单了。

好了，我已经给出了使用重载而不是事件处理器的所有理由。那 .NET 框架设计者确定有必要添加事件处理器么？当然有必要。除了我们，他们太忙以至于不会写没有人用的代码。重载只是在子类使用。其他的类都必须使用事件机制。这意味这在 XAML 文件中定义声明就可以使用事件处理器。在上面的例子中，你的设计者可能会有鼠标按下的事件的行为。设计者可以创建 XAML 声明它们的行为。这些行为可以响应表单事件。你可以在代码中重定义所有行为，但那会花更多时间处理一个事件。这只是将设计者的问题转移给你。你清楚自己想要设计者为你处理设计工作。更好的方式是创建一个事件然后通过设计工具创建 XMAL 声明。所以最后，你创建一个新的类并发送事件到表单类。这样一开始就添加一个事件处理器到表单会更简单。毕竟，这就是为什么 .NET 框架设计者要在表单添加事件。

事件机制的另外一个理由是事件是在运行时被链接的。使用事件会有个更大的灵活性。你可以根据程序的不同状态链接不同的事件处理器。假设你在写一个画图程序。根据程序的不同状态，鼠标按下可能开始画线，或者可能是选择一个对象。当使用者切换模式，你可以切换事件处理器。不同的类，不同的事件处理器，事件的处理依赖于程序的状态。

最后，使用事件，你可以在一个事件上挂多个事件处理器。再想象刚才的画图程序。你可能有多个事件处理器跟鼠标事件挂钩。第一个是具体的行为。第二个是更新状态栏或者更新不同命令的可用性。多个行为可以在同一个事件响应。

当你基类有一个函数处理一个事件，重载是更好的方法。这个更容易维护，s即使时间退役也更可能是正确的，并且更高效。保留事件处理器做其他用途。更倾向与重载基类的实现而不是附加事件处理器。

小结：

这个原则就是说 WPF 编程实现事件响应要选择重载虚函数的方式，而不是事件模式！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2087024

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则31：用 IComparable<T> 和 IComparer<T> 实现排序关系

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

你的类型需要排序关系用于定义集合的排序和搜索。 .NET 框架定义两个接口描述类型的排序关系： IComparable<T> 和 IComparer<T> 。IComparable 定义了类型的自然排序。类型实现的 IComparer 描述的另一个排序。你可以在接口中提供具体类型的比较并避免运行时低效率的自定义关系操作符（ < ， >， <=， >= ）的实现。本原则会讨论怎么实现排序关系，.NET 核心框架可以根据你定义的接口对你的类型进行排序，其他使用者可以这些操作得到更好的性能。

IComparable 接口包含一个方法： CompareTo() 。这个方法遵循了从 C 库函数 strmp 开始的传统：它返回小于0的数如果当前对象小于比较的对象，如果它们相等返回0，如果当前对象比比较的对象大则返回大于0的数。 IComparable<T> 会被 .NET 框架的大多数新 APIs 使用。然而，一些旧的 APIs 仍使用旧的的 IComparable 接口。因此，当你实现 IComparable<T> ， 你也应该实现 IComparable 。IComparable 的参数类型为 System.Object 。你需要在运行时对参数进行检查。每次比较都是耗时的，你必须重新解读参数的类型：

```
 和 IComparer <t>实现排序关系" style="display: none;">public struct Customer : IComparable<(
{
    private readonly string name;
    public Customer(string name)
    {
        this.name = name;
    }
    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion
    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException("Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion
}</t>
```

注意到这个结构体显示实现了 IComparable 。这就保证调用之前的接口实现参数为 Object 类型版本的 CompareTo() 的代码。太不喜欢旧版本的 IComparable 。你不得不要对参数的运行时类型进行检查。不正确的代码可以合法调用使用任何参数调用 CompareTo 方法。更糟糕的是，合适的参数也必须经过封箱和拆箱来进行比较。这样每次比较多了额外的运行花费。对

集合进行排序，使用 IComparable 平均进行了 nlg(n) 次比较。每次都会引起三次的封箱和拆箱操作。对于长度为1000的数组，会有20000次封箱和拆箱操作，平均：nlg(n) 大约为7000，并且每次比较有3次封箱和拆箱的操作。

你可能会奇怪为什么你需要实现非泛型的 IComarable 接口。有两个理由。首先，这只是简单的向后兼容。你的类型交互的代码在 .NET 2.0之前创建的。这意味着支持泛型之前的接口。第二，即使现代的代码都会避免使用泛型，当它是基于反射。反射可能使用泛型，但是这比没有泛型定义的反射困难得多。支持非泛型版本的 IComparable 接口使得你的类型很容易使用利用反射的算法。

因为旧版本的 IComparable.CompareTo() 已经被接口显式实现，它只能通过 IComparable 引用调用。你 Customer struct 的使用者可以获得类型安排的比较，但类型不安全的比较是不可访问的。下面无辜的错误是不会通过编译：

```
 和 IComparer <t>实现排序关系" style="display: none;">Customer c1;
Employee e1;
if (c1.CompareTo(e1) > 0)
    Console.WriteLine("Customer one is greater");</t>
```

这不能通过编译因为参数对于 public Customer.CompareTo(Customer right) 方法是错误的。IComparable.CompareTo(object right) 方法是不可访问的。你只能通过显式类型转换才能访问 IComparable 方法：

```
 和 IComparer <t>实现排序关系" style="display: none;">Customer c1 = new Customer();
Employee e1 = new Employee();
if ((c1 as IComparable).CompareTo(e1) > 0)
    Console.WriteLine("Customer one is greater");</t>
```

当你实现 IComparable，使用显式的接口实现并提供一个强类型的 public 重写。强类型的重写提供性能而且减少一些人错误使用 CompareTo 方法的概率。你没有看到 .NET 框架使用 Sort 函数的所有优势，因为它通过接口指针访问 CompareTo() 方法（查看原则22），但是代码知道两个比较对象的类型会有更好的性能。

我们对 Customer struct 进行最后的小改变。C# 语言运行你重写标准关系操作符。这些可以充分利用类型安全的 CompareTo() 方法：

```
 和 IComparer <t>实现排序关系" style="display: none;">public struct Customer : IComparable<
{
    private readonly string name;
    public Customer(string name)
    {
        this.name = name;
    }
    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion
    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
        throw new ArgumentException("Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion
    // Relational Operators.
    public static bool operator <(Customer left, Customer right)
    {
        return left.CompareTo(right) < 0;
    }
    public static bool operator <=(Customer left, Customer right)
    {
        return left.CompareTo(right) <= 0;
    }
    public static bool operator >(Customer left, Customer right)
    {
        return left.CompareTo(right) > 0;
    }
    public static bool operator >=(Customer left, Customer right)
    {
        return left.CompareTo(right) >= 0;
    }
}</t>
```

这就是标准的 Customer 的排序：通过名字。后面你必须创建对所有消费者根据收入的排序。你还是需要定义在 Customer struct 内的正轨的比较函数，即通过名字进行排序。在泛型成为 .NET 框架一部分后，很多 APIs 开发都会要求执行其他排序的 Comparison<T> 的委托。在 Customer 类中很简单创建一个执行其他排序的静态属性。例如，下面这个委托比较两个消费者的收入:

```
 和 IComparer <t>实现排序关系" style="display: none;">public static Comparison<Customer> Co
{
    get
    {
    return (left,right) =>
        left.revenue.CompareTo(right.revenue);
    }
}</t>
```

旧的类库会要求使用 IComparer 接口的类似功能。 IComparer 提供可选的标准没有泛型的排序。任何发布在 1.x .NET FCL 的 IComparable 实现方法都提供通过 IComparer 排序的重写。因为你是 Customer struct 的作者，你可以为次在 Customer struct 内部创建一个 pirvate

嵌套类（RevenueComparer）。它会暴露 Customer strcut 的一个静态属性：

```
 和 IComparer <t>实现排序关系" style="display: none;">public struct Customer : IComparable<
{
    private readonly string name;
    private double revenue;
    public Customer(string name, double revenue)
    {
        this.name = name;
        this.revenue = revenue;
    }
    #region IComparable<Customer> Members
    public int CompareTo(Customer other)
    {
        return name.CompareTo(other.name);
    }
    #endregion
    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
        throw new ArgumentException("Argument is not a Customer", "obj");
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
    #endregion
    // Relational Operators.
    public static bool operator <(Customer left, Customer right)
    {
        return left.CompareTo(right) < 0;
    }
    public static bool operator <=(Customer left, Customer right)
    {
        return left.CompareTo(right) <= 0;
    }
    public static bool operator >(Customer left, Customer right)
    {
        return left.CompareTo(right) > 0;
    }
    public static bool operator >=(Customer left, Customer right)
    {
        return left.CompareTo(right) >= 0;
    }
    private static RevenueComparer revComp = null;
    // return an object that implements IComparer
    // use lazy evaluation to create just one.
    public static IComparer<Customer> RevenueCompare
    {
        get
        {
            if (revComp == null)
                revComp = new RevenueComparer();
            return revComp;
        }
    }
    public static Comparison<Customer> CompareByReview
    {
        get
        {
        return (left,right) =>
            left.revenue.CompareTo(right.revenue);
        }
    }
    // Class to compare customers by revenue.
    // This is always used via the interface pointer,
    // so only provide the interface override.
    private class RevenueComparer : IComparer<Customer>
    {
        #region IComparer<Customer> Members
        int IComparer<Customer>.Compare(Customer left, Customer right)
```

```
        {
            return left.revenue.CompareTo(right.revenue);
        }
        #endregion
    }
}</t>
```

最后的 Customer struct 版本，内嵌了 RevenueComparer ，你可以根据名字顺序，自然顺序进行排序，而且提供根据消费者收入排序实现的 IComparer 接口的可选排序。如果你不能访问 Customer 类的代码，你可以使用 public IComparer 的属性对消费者进行排序。当你不能访问到类的代码，就好像你对 .NET 框架的类使用不同的排序，你应该使用这种用法。

在本原则中我还没有提到 Equals() 或 == 操作符（查看原则6）。排序关系和相等是不同的操作。你不需要在排序关系中实现相等比较。事实上，引用类型通常根据对象内容实现排序，而相等是基于对象是否是同一个对象来判断。 CompareTo() 返回0，即使 Equals() 返回 false。这是完全合法的。相等和排序关系没有必要相同。

IComparable 和 IComparer 是为你类型提供排序关系的标准机制。IComparable 主要用在自然排序中。当你实现 IComparable ，你也应该重写和 IComparable 排序一直的比较操作符（< ， > ， <= ， >= ）。 IComparable.CompareTo() 使用的是 System.Object 参数，所以你还应该提供具体类型的 CompareTo() 方法的重写。 IComparer 可以提供另外的排序或者当类型没有为你提供的排序。

小结：

每天只能发原创博客两篇，先占个位置，未完待续！

终于整完这篇了，今天刚好是下半年的第一天（虽然现在已经是7月2日的凌晨1:20了），上半年因为身体状态不行，就一直耽搁了，虽然我要更加努力！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2087383

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则32：避免 ICloneable

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

IConeable 听起来是一个很不错的想法：你类型实现了 IConeable 接口然后就支持复制。如果你不想要支持复制，你就不需要实现它。但是你的类型不能在真空中存在。你支持 IConeable 的决定会影响它的子类。一旦一个类型支持 ICloneable ，它的所有子类也都必须支持 ICloneable 。它的所有成员的类型都必须支持 ICloneable 或者有其他机制去复制。最后，如果支持深度复制并且当你类型包含 web 对象就会很有问题。 ICloneable 的官方定义就给出了这个问题：它支持深复杂或浅复制。浅复制创建一个新对象包含所有成员变量的复制。如果这些成员变量是引用类型的，新对象引用和原来对象是同一个对象。深复制创建新对象同样包含所有成员变量的复制。所有的引用类型都会被嵌套复制。对于内置变量，例如整数，深复制和浅复制产生的结果是一样的。类型支持哪一个？这就依赖于具体的类型。但是在同一个对象混合深复制和浅复制会引起相当不一致的表现。当你去趟了 ICloneable 的浑水，这就再说难免了。大多数情况下，避免使用 ICloneable 使得类更简单。它很容易使用，并且很容易实现。

任何只包含内置类的成员变量的值类型不需要支持 ICloneable ；简单的赋值复制 struct 的所有值比 Clone() 更高效。Clone() 会对返回值进行封箱，以至于强制转换为 System.Object 的引用。调用者必须进行强制类型转换才能从箱中提取值。这样做已经够了。不要重写 Clone() 函数来进行赋值复制。

如果值类型包含引用类型会怎么样？最常见的例子是值类型包含一个 string :

```
public struct ErrorMessage
{
    private int errCode;
    private int details;
    private string msg;
    // details elided
}
```

string 是一个特殊例子因为它是不可变的类。如果你赋值一个 ErrorMessage 对象，两个 ErrorMessage 对象会引用相同一个字符串。它不会引起一般引用类型的可能会出现的错误。如果你通过任何一个引用改变 msg 变量，会创建一个 string 对象（查看原则16）。

普遍的例子是创建一个包含任意引用变量的 struct 会更复杂。这也很少见。struct 内置的赋值创建浅复制，两个 struct 会引用相同的对象。为了创建深复制，你需要克隆包含的引用类型对象，而且你需要知道这个引用类型通过 Clone() 方法来支持深复制。这样，要做的工作如果包含的引用类型支持 ICloneable ，并且它的 Clone() 方法创建深复制。

下面我们开始讨论引用类型。引用类型支持 ICloneable 接口说明它们支持浅复制或深复制。你应该谨慎支持 ICloneable 因为这样就必须让这个类的所有子类也支持 ICloneable 。考虑下面的简单的继承结构：

```
class BaseType : ICloneable
{
    private string label = "class name";
    private int[] values = new int[10];
    public object Clone()
    {
        BaseType rVal = new BaseType();
        rVal.label = label;
        for (int i = 0; i < values.Length; i++)
            rVal.values[i] = values[i];
        return rVal;
    }
}
class Derived : BaseType
{
    private double[] dValues = new double[10];
    static void Main(string[] args)
    {
        Derived d = new Derived();
        Derived d2 = d.Clone() as Derived;
        if (d2 == null)
            Console.WriteLine("null");
    }
}
```

如果你运行这个程序，你会发现 d2 的值是 null 。Derived 类从基类 BaseType 继承 ICloneable.Clone() ,但是实现却对子类是不正确的：它只是克隆基类 BaseType.Clone() 创建基类对象，而不是子类对象。这就是测试程序中为什么 d2 为 null —— 它不是 Derived 对象。然而，即使你克服了这个问题， BaseType.Clone() 不能复制定义在 Derived 的 dValues 数组。所以当你实现 ICloneable ，你必须强制所有子类也都实现。实际上，你可以提供一个钩子函数让所有子类能有自己的实现（查看原则23）。为了支持克隆，子类只能添加实现了 ICloneable 的值类型或引用类型的成员变量。这是对于子类是非常严格的限制。在基类支持 ICloneable 增加了子类的负担，所以你应该在非封闭的类避免实现 ICloneable 。

如果整个类的继承结构都必须实现 ICloneable ， 你可以差un感觉一个 abstract Clone() 方法，强制子类实现它。在这些例子，你还需要定义子类复制基类成员的方法。可以定义一个 protected 的复制构造函数：

```
class BaseType
{
    private string label;
    private int[] values;
    protected BaseType()
    {
        label = "class name";
        values = new int[10];
    }
    // Used by devived values to clone
    protected BaseType(BaseType right)
    {
        label = right.label;
        values = right.values.Clone() as int[];
    }
}
sealed class Derived : BaseType, ICloneable
{
    private double[] dValues = new double[10];
    public Derived()
    {
        dValues = new double[10];
    }
    // Construct a copy
    // using the base class copy ctor
    private Derived(Derived right) : base(right)
    {
        dValues = right.dValues.Clone() as double[];
    }
    public object Clone()
    {
        Derived rVal = new Derived(this);
        return rVal;
    }
}
```

基类没有实现 ICloneable ；提供了 protected 的复制构造函数，让子类能拿复制基类的部分。叶节点的类都是封闭的，当有必要的时候实现 ICloneable 。基类不会强制所有子类实现 ICloneable ， 但是必须提供子类因支持 ICloneable 而需要的方法。

ICloneable 仍有它的用处，但是这是一个例外而不是指导规则。 .NET 框架更新支持泛型时，而没有添加 ICloneable<T> 的支持是非常有意义的。你不应该对值类型添加 ICloneable 的支持；而是使用赋值操作。当复制操作对叶借点封闭类很重要，你就应该添加 ICloneable 支持。当基类支持 ICloneable 你就为此创建 protected 复制够函数。对于其他的所有情况，避免使用 ICloneable 。

小结：

这个原则其实强调的重点是不管是值类型还是引用类型如果实现了 ICloneable 接口，这个类的成员变量和继承结构也要实现 ICloneable ， 才能做到深复制和浅复制的一致性。这点其实跟 Java 是一样的！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

原则33：只有基类更新处理才使用 new 修饰符

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

你可以使用 new 修饰符重定义从基类继承的非虚成员。仅仅是因为你可以做某些事情却不意味着你应该做。重定义非虚方法会有歧义的行为。大多数开发者看到下面两块代码会立即认为他们执行的是同一件事情，如果它们继承于同一个对象：

当涉及到 new 修饰符，就不是这样的情况：

```
public class MyClass
{
    public void MagicMethod()
    {
        // details elided.
    }
}
public class MyOtherClass : MyClass
{
    // Redefine MagicMethod for this class.
    public new void MagicMethod()
    {
        // details elided
    }
}
```

这类实践会困扰很多开发者。如果你对同一个对象调用同一个函数，你会期望相同的代码被执行。实际上改变引用，即类名标签，你调用函数改变了行为给人错误的感觉。这是不一致的。MyOtherClass 对象的行为取决于你怎么引用它。实际 new 修饰符的修饰不会使非虚方法变成虚方法。而是，它在不同类名作用域增加了不同的方法。

非虚方法是静态绑定的。任何地方的引用 MyClass.MagicMethod() 的代码调用都执行这个函数。运行时不会去查找子类定义的不同版本。虚函数，从另一方面讲，是动态绑定的。运行时会根据对象的运行时类型调用真正的函数。

推荐避免使用 new 修饰符去重定义非虚函数并不是说建议你把基类的函数都定义为虚的。类库设计者定义虚函数是有一定的讲究。你定义虚函数说明任何子类期望改变虚函数的实现。虚函数集定义了子类期望改变的所有行为。“默认为虚”设计是说子类可以改变基类的所有行为。这确实是说你不会考虑子类修改的行为的所有后果。相反，你应该花时间去思考哪些方法和属性声明为多态。使那些并且只有那些是虚的。不要认为这可以限制你的类型的使用。而是，要把它作为提供自定义你的类型的行为的入口。

有一个地方，只有一地方，你要使用 new 修饰符。你在基类已经使用的方法签名使用 new 修饰符就构成新的版本。你获得的函数代码取决于你类的函数名字。可能有其他程序集使用你的方法。你在你的类库创建下面类，使用 BaseWidget 定义在其他类库中：

```
public class MyWidget : BaseWidget
{
    public void NormalizeValues()
    {
        // details elided.
    }
}
```

你完成你的组件，并且客户正使用它。而后你发现 BaseWidget 公司发布了新版本。渴望得到新的特性，你理解购买她并尝试构建 MyWidget 类。就会失败因为 BaseWidget 自己添加了 NormalizeValues 方法：

```
public class BaseWidget
{
    public void Normalizevalues()
    {
        // details elided.
    }
}
```

这就会有问题。你的基类偷偷在你的类名作用域下加了一个方法。有两种方式修复它。你可以改变你的类型 NormalizeValues 方法的名字。注意直观上认为 BaseWidget.NormalizeValues() 和 MyWidget.NormalizeValues() 在语意上的相同操作。如果不是，你不应该调用基类的实现。

```
public class MyWidget : BaseWidget
{
    public void NormalizeAllValues()
    {
        // details elided.
        // Call the base class only if (by luck)
        // the new method does the same operation.
        base.NormalizeValues();
    }
}
```

或者，你可以使用 new 修饰符：

```
public class MyWidget : BaseWidget
{
    public void new NormalizeValues()
    {
        // details elided.
        // Call the base class only if (by luck)
        // the new method does the same operation.
        base.NormalizeValues();
    }
}
```

如果你可以得到 MyWidget 类的所有客户端代码，你可以改变方法名字因为这很容易长期运行。但如果你是向世界发布 MyWidget 类，就会强制你的所有使用者做无数的更改。这就是 new 修饰符派上用场的地方。你的客户可以不更改继续使用 NormalizeValues() 方法。没有人

会调用 BaseWidget.NormalizeValues() 因为它在之前不存在。 new 修饰符可以处理这样的情况，即升级基类使得有一个成员和之前声明的类有冲突的情况。

当然，随着时间的推移，你的使用者可能需要使用 BaseWidget.NormalizeValues() 方法。那样你还是回到原来的问题：两个方法看起来相同但是表现却不同。考虑所有 new 修饰符的长期后果。有时，短期更改你的方法是不方便却更好。

new 修饰符必须谨慎使用。如果你不分青红皂白，你就在创建歧义的方法。它只用于这个在升级基类引起和你的类冲突的特例。即使是那样的情况，在使用它之前也要仔细考虑。最重要的是，不要在其他情况使用它。

小结：

尽量不用 new 修饰符，减少歧义方法。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2087773

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则34：避免定义在基类的方法的重写

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

当基类给成员命名时，名字就赋予了语义。在任何情况下，子类最好都不要将同一个名字用作其他目的。但为什么有很多其他原因子类会使用同一个名字。它可能想用不同方式实现同一个语义，或者有不同的参数。有时这是语言原生就支持：类设计者可以声明一个虚函数，子类就可以各自实现语义。原则33包含了为什么使用 new 修饰符 可能导致很难发现代码的 bug 。在这个原则里，你将会学到重写基类定义的函数会导致类似的问题。你应该不要重写基类声明的方法。

C# 语言重载解析的规则必然是非常复杂的。所有可能的子类声明的方法，基类的任何方法，以及扩展方法和实现接口的方法都是解析的候选方法。增加了泛型方法和泛型扩展方法，就变得更复杂。使用默认参数，而且我不确定每个人能否确切知道结果是什么。你真的想要这个情况更加复杂？创建的声明在基类函数的重写版本会增加找到最好函数匹配的复杂性。这也增加歧义的可能性。这也增加你的解释和编译器不同的可能性，就自然困扰使用者。解决方法是很简单的：给方法选择不同的名字。这是你的类，你就有足够的光彩去给方法提出一个不同名字，尤其当很容易困扰类的使用者时。

这个指导是直截了当的，然而总是有人会怀疑真的要这么严格。可能是因为重写听起来很像重载。重载虚函数是面向对象语言的核心原则；那明显不是我的意思。重写是使用不同的参数列表创建多个名字一样的方法。重写基类方法真的对重写解析有很多影响？我们从不同的方式看在重写基类中的方法会引起问题。

这个问题有很多组合情况。我们从最简单的开始。很多时候基类和子类更多的是不同参数列表的重写。字啊么所有的例子，任何基类类名以“B”开始，任何子类类名以“D”开始。下这个例子用这个继承关系的类作为参数：

```
public class B2 {}
public class D2 : B2 {}
```

下面这个类的方法使用子类（D2）作为参数：

```
public class B
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }
}
```

显然，这段代码会输出“In B.Foo”：

```
var obj1 = new D();
obj1.Bar(new D2());
```

下面，我们在子类中重写这个方法：

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }
}
```

那么，执行下面这段代码会发生什么？

```
var obj2 = new D();
obj2.Foo(new D2());
obj2.Foo(new B2());
```

两行都是输出“In D.Foo”。你调用的都是子类的方法。很多开开发者会认为第一个函数调用会输出“In B.Foo”。然而，即使很简单的重写都会很惊奇。两个调用都解析为 D.Foo 的原因是当有多个候选方法时，编译时继承关系最底端的子类的方法是最好的选择。即使当基类有更好的匹配这个规则都是正确的。当然，也是非常脆弱的。你认为下面的结果是什么：

```
B obj3 = new D();
obj3.Foo(new D2());
```

上面我的用词非常小心因为 obj3 的编译时类型为 B （你的基类），即使它的运行时类型是 D （你的子类）。 Foo 不是虚函数， obj3.Foo() 一定会被解析为 B.Foo 。

如果你的使用者基础差并想要解析规则跟他们期望的一样，他们就需要使用强制类型转换：

```
var obj4 = new D();
((B)obj4).Foo(new D2());
obj4.Foo(new B2());
```

如果你的 API 强制你的使用者这样构造，你就会遇到很多挫折。你还可以很容易增加一点困扰。在你的基类 B 增加一个方法：

```
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }
    public void Bar(B2 parm)
    {
        Console.WriteLine("In B.Bar");
    }
}
```

毫无疑问，下面的代码会打印出“In B.Bar”：

```
var obj1 = new D();
obj1.Bar(new D2());
```

现在，增加另一种重写，包含一个默认参数：

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }
    public void Bar(B2 parm1, B2 parm2 = null)
    {
        Console.WriteLine("In D.Bar");
    }
}
```

希望，你已经看到将会发生什么。同样的代码现在会打印出“In D.Bar”（你又调用子类）：

```
var obj1 = new D();
obj1.Bar(new D2());
```

唯一的调用基类的方法的方式是在调用代码中提供强制类型转换。

这几个例子展示一个参数的方法会遇到的问题。如果你的参数是基于泛型的会变得越来越复杂。假设你增加下面的方法：

```
public class B
{
    public void Foo(D2 parm)
    {
        Console.WriteLine("In B.Foo");
    }
    public void Bar(B2 parm)
    {
        Console.WriteLine("In B.Bar");
    }
    public void Foo2(IEnumerable<D2> parm)
    {
        Console.WriteLine("In B.Foo2");
    }
}
```

进而，在子类添加不同的重写：

```
public class D : B
{
    public void Foo(B2 parm)
    {
        Console.WriteLine("In D.Foo");
    }
    public void Bar(B2 parm1, B2 parm2 = null)
    {
        Console.WriteLine("In D.Bar");
    }
    public void Foo2(IEnumerable<B2> parm)
    {
        Console.WriteLine("In D.Foo2");
    }
}
```

按照前面的方式调用 Foo2 ：

```
var sequence = new List<D2> { new D2(), new D2() };
var obj2 = new D();
obj2.Foo2(sequence);
```

这回你会认为输出什么？如果你花了心思，你会发现“ In D.Foo2 ”会被输出。你会这个答案半信半疑。这个就是 C# 4.0 的变化。从 C# 4.0 开始，泛型接口支持协变和逆变，这意味 D.Foo2 是参数为 IEnumerable<D2> 的候选方法，尽管它的参数类型是 IEnumerable<B2> 。然后，更早的 C# 版本泛型不具有可变性。也就是说泛型参数是不可变的。在那些版本，当参数为 IEnumerable<D2> 时，D.Foo2 就不是候选方法。唯一的的候选方法是 B.Foo2 ，在那些版本中它是正确的答案。

上面的代码表明，有时很多复杂的情况你需要强制类型转换帮助编译器选择你想要的方法。在现实世界中，毫无疑问你会遇到需要使用强制类型转换而不是靠编译器选择“最好”的方法的情况，因为类继承关系，实现接口和扩展方法一起组成你想要的方法。但事实上，现实世界丑陋的情况偶尔才发生不意味着你创建更多的重写方法给自己增加更多问题。

现在你就可以在程序员的鸡尾酒会因拥有更深入 C# 重载解析而让你朋友震惊。这很有用的信息，并且你对语言了解的更多，你就会是更好的开发者。但是不要期望你的使用者会有和你一样层次的知识。更重要的是，不要以为使用你的 API 的每个人都对重写解析怎么工作的都有详细的掌握。而是，不要重写在基类声明的方法。那样不会提供任何作用，并且只能是导致你的使用者的困扰。

小结：

记住，重写解析规则是优先选择编译时继承结构最底端的子类的方法，即使基类有更匹配的方法，C# 4.0以后版本泛型也遵循这个规则，避免重写，这则原则的要义就掌握了！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2088635

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则35：理解 PLINQ 并行算法的实现

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

在这个原则我希望我能说并行编程现在和在你循环添加 AsParall() 一样简单。虽然不是，但是 PLINQ 使得更容易利用多核并且程序仍然是正确的。创建多和程序绝不是微不足道，但 PLINQ 使得它更简单。

你已经理解访问数据必须是同步的。你仍需要衡量下声明在 ParallEnumerable 的并行和串行版本的方法的影响。LINQ 涉及的一些方法很容易就可以并行执行。其他要强制串行访问序列的每个元素——或者，至少，需要完整的序列（比如 Sort ）。让我们通过一些使用 PLINQ 的例子学习是在怎么工作的，并且那些地方还存在缺陷。这节的所有例子和讨论都是对 Object 使用 LINQ 。标签甚至可以叫做 “ Enumerable ” 而不是“ Queryable ”。 PLINQ 不会帮助你 SQL 的 LINQ ，或者是实体框架的算法的并行执行。这不是一个真正的限制特性，因为这些的实现利用并行数据库引擎来执行并行查询。

这个例子是很简单的查询，使用语法方法调用对前150个计算 n! ：

```
var nums = data.Where(m => m < 150). Select(n => Factorial(n));
```

你在查询第一个函数调用增加 AsParallel() 使得查询是并行的：

```
var numsParallel = data.AsParallel().Where(m => m < 150).Select(n => Factorial(n));
```

当然，你可以使用查询语法做类似的工作。

```
var nums = from n in data where n < 150 select Factorial(n);
```

并行的版本依赖于 data 序列增加的 AsParallel() ：

```
var numsParallel = from n in data.AsParallel() where n < 150 select Factorial(n);
```

这个结果是调用方法的版本是一样的。

第一个例子很简单，但通过你调用的方法 PLINQ.AsParallel() 选择并行执行任何查询表达式说了几个重要的概念。一旦你调用 AsParallel() ，在多核就是使用 多线程 Thread.AsParallel() 分成子序列操作进行并且返回 IParallelEnumerable() 而不是 IEnumerable() 。 PLINQ 就是 IParallelEnumerable 实现的扩展方法集。它们大多数都和扩展 IEnumerable 的 Enumerable

类的扩展方法有一样的签名。IParallelEnumerable 只是简单替换 Enumerable 的参数和返回值。这样做的优势是 PLINQ 遵循所有 LINQ 遵循的模式。这使得 PLINQ 非常容易学习。你对 LINQ 的任何掌握，通常，都可以应用于 PLINQ 。

当然，这也不是那么简单。初始查询很容就能应用 PLINQ 。它没有任何共享数据。结果的次序没有任何影响。这就是为什么代码运行获得的加速直接和机器的核心数成正比。为了帮助 PLINQ 获得最好的性能，IParallelEnumerable 有很多控制并行任务类库函数。

每个并行查询都是从分区步骤开始。 PLINQ 需要对输入元素进行分区并分配它们一些任务去执行查询。分区是 PLINQ 最重要的一方面，所以理解 PLINQ 不同的方法， PLINQ 决定使用哪个方法，和每个方法都是如何工作的是非常重要的。首先，分区不能花太多时间。如果 PLINQ 类库花费太多时间在分区上，那么留给处理数据的时间就太少了。 PLINQ 会根据输入源和创建的查询类型决定使用不同的分区算法。最简单分区算法是范围分区。范围分区会根据任务数量划分输入序列并将每部分分配给一个任务。例如，1000个元素的序列运行在四核的机器上会被分为每250的四部分。范围分区只有当查询数据源支持索引序列和报告了序列的长度才会被使用。也就是说范围分区被限制在像 List<T> 数组，和其他支持 IList<T> 的序列的查询数据源。当查询数据源支持这些操作范围分区经常被使用。

第二个分区选择是块分区。这个算法给每个任务分配一个输入元素块，并且这会需要更多功夫。内部的块算法会与时俱进，所以我不会深入涉及当前的实现。你会认为块的开始会很小，因为输入源可能很小。这样就可以防止一个任务处理整个小序列。你也会认为随着工作的持续，块的大小会变大。这可以最小化线程的开销和有助于最大化吞吐量。块也有可能根据查询委托和 where 子句过滤的元素花费的时间调整大小。这样的目的是所有的任务能在接近的时间完成，也就是最大化整体的吞吐量。

另外两个分区方案会优化某些查询操作。第一个是带分区。带分区是范围分区的特例，它优化处理序列开始的元素。每个工作线程都跳过前面N项然后处理后面的M项。在处理M项之后，工作线程会接着跳过下N项。带算法很容易理解，假设带宽为1个元素。在四个工作任务的情况下，第一个任务获得项的下标为0，4，8，12等等，第二个任务获得项下标为1，5，9，13等等。整个查询过程，带算法避免任何线程内部的同步实现 TakeWhile() 和 SkipWhile() 。同时，每个工作线程移到下一个元素只需要很简单算术运算。

最后的算法是哈希分区。哈希分区是针对 Join ， GroupJoin ， GroupBy ， Distinct ， Except ， Union ， 和 Intersect 操作的查询设计的特殊目标的算法。这些耗时的操作，特定的分区算法可以使得这些查询更大的并行。Hash 算法保证所有产生的相同的哈希码被同一个任务处理。这就最小化任务之间的处理这些操作的交流。

除了分区算法， PLINQ 还是用三个不同的算法并行你的代码的任务： Pipelining ， Stop & Go 以及 Inverted Enumeration 。 Pipelining 是默认的，所以我首先进行解释。 Pipelining 算法，一个线程处理枚举（ foreach 或者 查询序列）。多个线程用来处理处理每个元素的查询。每个请求的新元素，都会在其他线程被处理。 PLINQ 在 Pipelining 模式使用的线程数经常是核心的数量（对于大多基于 CPU 查询）。在我的阶乘的例子，它在我的双核机器上使用两个线程。第一个元素被检索并在一个线程处理。紧接着第二个元素会被请求并在第二个线

程处理。然后，当这两个有一个完成，第三个元素就会被请求，并且查询表达式会在这个线程处理。整个序列查询的执行过程中，两个线程都忙于元素的查询。机器愈多核，更多元素会被并行处理。

例如，在16核的机器上，前16项会在16不同线程上立即处理（推测运行在16个不同核心上）。我已经进行简化了，有一个线程处理枚举，这就说明通常 Pipelining 创建（核心数+1）的线程。在大多数情况下，枚举线程会等很长时间，所以创建额外线程是有意义的。

Stop & Go 算法意味着枚举的线程会加入到其他线程运行查询表达式。当你需要立即执行 ToList() 或 ToArray() 和任何时候 PLINQ 排序之前需要所有结果集等查询时，这个算法会使用。下面两个查询都使用 Stop & Go 算法：

```
var stopAndGoArray = (from n in data.AsParallel()
    where n < 150
    select Factorial(n)).ToArray();
var stopAndGoList = (from n in data.AsParallel()
    where n < 150
    select Factorial(n)).ToList();
```

使用 Stop & Go 算法处理会占用更多内存并获得更好的性能。然而，注意我在进行任何查询表达式之前已经构造整个查询。你要构成整个查询而不是处理每部分都使用 Stop & Go 算法然后在将最后的结果组合起来进行其他查询。这往往会引起引起线程开销而降低性能。像一个组合操作一样处理整个查询表达式总是一个更好的选择。

并行任务库使用的最后一个算法是倒计数算法。 Inverted Enumeration 不会产生任何结果集。而是每个查询的结果执行某些行为。在我前面的例子章，我向控制台输出阶乘计算的结果：

```
var numsParallel = from n in data.AsParallel()
    where n < 150
    select Factorial(n);
foreach (var item in numsParallel)
    Console.WriteLine(item);
```

Object 的 LINQ （非并行）被认为是非常懒的。也就是只有请求了才会产生值。当你处理查询结果时，可以选择并行执行模型。这就是为什么你会需要 Inverted Enumeration 模型：

```
var nums2 = from n in data.AsParallel()
    where n < 150
    select Factorial(n);
nums2.ForAll(item => Console.WriteLine(item));
```

Inverted Enumeration 比 Stop & Go 方法使用更少内存。同时，它可以在结果集上并行操作。注意你在 ForAll() 查询仍需要使用 AsParallel() 。 ForAll() 比在 Stop & Go 模型占用更少内存。在某些情况，依赖于查询表达式的结果集上的行为的工作量，通常 Inverted Enumeration 会并枚举方法更快。

所有的 LINQ 查询的执行都很懒。你创建的查询，这些查询只有在请求查询结果的项时才执行。对于 Object 的 LINQ 有改进。Object 的 LINQ 在你需求元素时才对每个元素执行查询。PINQ 进行不同的工作。它们的模型跟 SQL 的 LINQ ，或者实体框架很接近。在这些模型，当你请求第一个元素，整个结果序列都会产生。 PLINQ 跟这些模型很接近，但又不全相同。如果你对 PLINQ 怎样执行查询有误解，你会使用超过必须的资源，这样你实际的并行查询运行在多核机器上比 Object 的 LINQ 更慢。

为了演示一些区别，我针对给出简单的查询。我会给你展示增加的 AsParallel() 怎样改变执行模型。两个模型都是有效的。 LINQ 关注的它的结果，而不是它们是如何产生的。你将看到两个模型都产生一样的结果。如果你算法对查询自己有因为区别就会显现出来。

下面的查询用来演示区别：

```
var answers = from n in Enumerable.Range(0, 300)
    where n.SomeTest()
    select n.SomeProjection();
```

我会输出显示 SomeTest() 和 SomeProjection() 方法的调用：

```
public static bool SomeTest(this int inputValue)
{
    Console.WriteLine("testing element: {0}", inputValue);
    return inputValue % 10 == 0;
}
public static string SomeProjection(this int input)
{
    Console.WriteLine("projecting an element: {0}", input);
    return string.Format("Delivered {0} at {1}",input.ToString(), DateTime.Now.ToLongTime
}
```

最后，用一个简单的循环，我使用 IEnumerator<string> 成员对结果进行遍历，可以看到不同行为的发生。这就对序列的产生（并行）和枚举（在枚举循环）显示的更清晰。在生成的代码，我更喜欢不同的实现。

```
var iter = answers.GetEnumerator();
Console.WriteLine("About to start iterating");
while (iter.MoveNext())
{
    Console.WriteLine("called MoveNext");
    Console.WriteLine(iter.Current);
}
```

使用标准的 Object 的 LINQ 实现，你会看到像下面的输出：

```
About to start iterating
testing element: 0
projecting an element: 0
called MoveNext
Delivered 0 at 1:46:08 PM
testing element: 1
testing element: 2
testing element: 3
testing element: 4
testing element: 5
testing element: 6
testing element: 7
testing element: 8
testing element: 9
testing element: 10
projecting an element: 10
called MoveNext
Delivered 10 at 1:46:08 PM
testing element: 11
testing element: 12
testing element: 13
testing element: 14
testing element: 15
testing element: 16
testing element: 17
testing element: 18
testing element: 19
testing element: 20
projecting an element: 20
called MoveNext
Delivered 20 at 1:46:08 PM
testing element: 21
testing element: 22
testing element: 23
testing element: 24
testing element: 25
testing element: 26
testing element: 27
testing element: 28
testing element: 29
testing element: 30
projecting an element: 30
testing element: 10
projecting an element: 10
called MoveNext
Delivered 10 at 1:46:08 PM
```

查询知道第一次枚举的 MoveNext() 的调用开始执行。第一次 MoveNext() 的调用查询了足够的元素检索出第一个结果序列上的元素（恰好是查询的第一个元素）。第二个 MoveNext() 处理输入序列的元素知道下一个输出元素的产生。使用 Object 的 LINQ， 每次调用 MoveNext 执行的查询直到下一个输出元素的产生。

要是你将查询改为并行查询，规则就会改变：

```
var answers = from n in ParallelEnumerable.Range(0, 300)
    where n.SomeTest()
    select n.SomeProjection();
```

这个查询的输出看起来非常的不一样。下面是运行一次的采样（每次运行可能有些不同）：

```
About to start iterating
testing element: 150
```

```
projecting an element: 150
testing element: 0
testing element: 151
projecting an element: 0
testing element: 1
testing element: 2
testing element: 3
testing element: 4
testing element: 5
testing element: 6
testing element: 7
testing element: 8
testing element: 9
testing element: 10
projecting an element: 10
testing element: 11
testing element: 12
testing element: 13
testing element: 14
testing element: 15
testing element: 16
testing element: 17
testing element: 18
testing element: 19
testing element: 152
testing element: 153
testing element: 154
testing element: 155
testing element: 156
testing element: 157
testing element: 20
... Lots more here elided ...
testing element: 286
testing element: 287
testing element: 288
testing element: 289
testing element: 290
Delivered 130 at 1:50:39 PM
called MoveNext
Delivered 140 at 1:50:39 PM
projecting an element: 290
testing element: 291
testing element: 292
testing element: 293
testing element: 294
testing element: 295
testing element: 296
testing element: 297
testing element: 298
testing element: 299
called MoveNext
Delivered 150 at 1:50:39 PM
called MoveNext
Delivered 160 at 1:50:39 PM
called MoveNext
Delivered 170 at 1:50:39 PM
called MoveNext
Delivered 180 at 1:50:39 PM
called MoveNext
Delivered 190 at 1:50:39 PM
called MoveNext
Delivered 200 at 1:50:39 PM
called MoveNext
Delivered 210 at 1:50:39 PM
called MoveNext
Delivered 220 at 1:50:39 PM
called MoveNext
Delivered 230 at 1:50:39 PM
called MoveNext
Delivered 240 at 1:50:39 PM
called MoveNext
Delivered 250 at 1:50:39 PM
```

```
called MoveNext
Delivered 260 at 1:50:39 PM
called MoveNext
Delivered 270 at 1:50:39 PM
called MoveNext
Delivered 280 at 1:50:39 PM
called MoveNext
Delivered 290 at 1:50:39 PM
```

注意到多大的改变了吧。第一次 MoveNext() 的调用使得 PLINQ 启动所有参与的线程产生结果。这过程产生更少（在这个例子，几乎所有的）结果对象。后续每次 MoveNext() 的调用都是抓取下来的项都是已经产生好的。你不能断定具体的输入元素什么时候会被处理。你只）知道的是当你请求查询的第一个元素是查询就开始执行（在几个线程

上）。

PLINQ 的方法理解查询语法的行为并影响查询的执行。假设你修改查询使用 Skip() 和 Take() 选择第二页的结果：

```
var answers = (from n in ParallelEnumerable.Range(0, 300)
    where n.SomeTest()
    select n.SomeProjection()).
    Skip(20).Take(20);
```

这个查询的执行结果和 Object 的 LINQ 的是相同的。这是因为 PLINQ 知道产生20个元素比300个更快。（我已经简化了，但是 PLINQ 的 Skip() 和 Take() 的实现更倾向于一个连续的算法而不是其他算法）。

你可以对查询再修改一点，而且 PLINQ 还是使用并行执行模型产生所有元素。只是增加 orderby 子句：

```
var answers = (from n in ParallelEnumerable.Range(0, 300)
    where n.SomeTest()
    orderby n.ToString().Length
    select n.SomeProjection()).
    Skip(20).Take(20);
```

orderby 的 lambda 参数一定不能被编译器优化的表达式（这就是为什么上面我使用 n.ToString().Length 而不是 n ）。现在，查询引擎必须在排序之前产生所有输出序列的元素。一旦元素被排序之后 Skip() 和 Take() 方法才知道哪些元素会被返回。当然在多核机器上使用多线程产生所有输出比顺序进行更快。 PINQ 也知道这点，所以它会启动多个线程来创建输出。

PLINQ 尝试创建你的写的查询的最好实现，产生你需要的结果的花费最少工作和最少时间。有时 PLINQ 查询会和你期望的不一样的方式执行。有时，它会表现的项 Object 的 LINQ ，请求输出序列的下一项才执行代码产生它。有时，它会的行为更像 SQL 的 LINQ 或实体框架即

请求第一个就会产生所有。有时它的行为更像两者的混合。你应该确保你不要引入 LINQ 查询的任何副作用。那些在 PLINQ 执行模型是不可靠的。你构建查询是需要确保你考虑大部分的底层技术。这就需要你理解它们的工作是怎样的不同。

并行算法被 Amdahl 法则限制：使用多处理器的程序被限制在程序的连续执行部分。ParallelEnumerable 的扩展方法对于这个规则毫无例外。很多方法都并行执行，但是有些因为它们的性质会影响并行的程度。显然 OrderBy 和 ThenBy 需要在任务之间进行协调。 Skip， SkipWhile， Take 和 TakeWhile 会影响并行程度。并行任务运行在不同的核心上完成的顺序可能不同。你可以使用 AsOrdered() 和 AsUnordered() 指示 PLINQ 是结果序列否受次序的影响。

有时你自己的算法会有副作用并不能并行。你可以使用扩展方法 ParallelEnumerable.AsSequential() 将并行序列解释为 IEnumerable 并强制顺序执行。

最后， ParallelEnumerable 包含允许你控制 PLINQ 执行并行查询的方法。你可以使用 WithExecutionMode() 建议并行执行，即使那会选择高开销的算法。默认情况下， PLINQ 会并行构造那些有并行需求的地方。你可以使用 WithDegreeOfParallelism() 建议在你的算法使用的线程数量。通常， PLINQ会分配和当前机器处理器数量一样多的线程。你可以使用 WithMergeOptions() 请求改变 PLINQ 查询过程中控制的的缓存结果。通常，PLINQ 会缓存每个线程的结果直到它们被消费者线程使用。你可以请求不缓存而是理解使用结果。你也可以请求权缓存，这会增加性能的高时延。默认的，自动缓存，很好的权衡时延和性能。缓存只是一个提示，而不是需求。PLINQ 可能忽略你的请求。

我没有给出这些设置的具体指导，因为最优方法高度依赖于你的算法。然而，你有这些设置可以改变，你可以在不同的目标机器上做实验查看是否会有利于你的算法。如果你没有多个不同的机器去实验，我建议你使用默认值。

PLINQ 使得并行计算比之前更简单。因为这些的增加时间会越来越重要；并行计算随着司空见惯的台式机和比较的核心数增加而越来越重要。这还不是很容易。设计糟糕的算法可能在并行上看不到性能的提高。你的任务就是反复查看这些算法并找出哪些是能并行的。尝试实现那些算法的并行版本。测试结果。和性能更好的算法一起工作。意识到一些算法很难实现并行，就让它们串行。

小结：

原创帖占位，未完待续！

工作中还没有怎么用到过 LINQ 和 PLINQ ，从本原则可以知道 PLINQ 可以实现并行，但是依赖于你的算法，所以既要对 PLINQ 有理解，也要能很好的设计你的算法才是最好的设计！

终于翻译完了（这个原则竟然有12页），太累了，坚持，坚持，再坚持！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2088663

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则36：理解 I/O 受限制（Bound）操作 PLINQ 的使用

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

Parallel Task 库看起来是 CPU 受限制操作进行优化。虽然这是类库的一个核心任务，但是它在 I/O 受限制操作也做的很好。实际上， Parallel Task 库的处理 I/O 受限制操作的设计更多是默认实现的。它会根据你线程的繁忙程度更新分配线程的数量。更多阻塞的线程（等待 I/O 操作）会导致线程分配更多线程完成手头的任务。

和其他并行扩展一样，你可以使用方法调用，或者 LINQ 查询语法进入并行执行模型。I/O 受限制操作和 CPU 受限制操作的并行执行表现会有些不同。你经常需要比核心更多的线程，因为 I/O 受限制操作花费更多时间等待它们的外部事件。PLINQ 也为这个习惯提供了框架。

下面这段代码是从一系列网站下载数据：

```
foreach (var url in urls)
{
    var result = new WebClient().DownloadData(url);
    UseResult(result);
}
```

DownloadData() 的调用发起了 web 的同步请求而且知道所有数据被检索下来。这个算法花费太多时间等待。你可以使用并行 for 循环改变为并行模型：

```
Parallel.ForEach(urls, url =>
{
    var result = new WebClient().DownloadData(url);
    UseResult(result);
});
```

Parallel.ForEach() 会选择进入并行处理模型。这个版本花的时间比顺序进行版本少多了。实际上，在我的双核机器上，加速和 url 的集合的元素个数成正比。线程花更多时间在等待，所以 Parallel Task 库会创建更多线程。

你可以使用 PLINQ 和查询语法产生一样的结果：

```
var results = from url in urls.AsParallel()
    select new WebClient().DownloadData(url);
results.ForAll(result => UseResult(result));
```

PLINQ 和 Parallel Task 库支持的 Parallel.ForEach() 会有一点不同。PLINQ 使用固定的线程数量，然而 AsParallel() 会更加吞吐量的增加和减少活动线程的数量。你可以使用 ParallelEnumeralbe.WithDegreeOfParallelism() 控制线程的数量（查看原则35）。但是 Parallel.ForEach() 会为你管理。 Parallel.ForEach() 当 I/O 受限制和 CPU 受限制操作混合时表现的更好。 Parrallel.ForEach() 会基于当前加载管理活动线程的数量。当更多线程阻塞等待 I/O 操作，它就会创建更多的线程以增加吞吐量。当工作线程增多，它就会让活动的线程数量减少以最小化上下文的切换。

上面演示的代码不是真正的异步。它只是充分利用多线程来并行执行些任务。但是围绕这个程序它会等待所有 web 请求完成才继续做其他工作。 Parallel Task 库提供其他基本的异步模式的实现。其中常见的模式之一就是启动多个 I/O 受限制任务并且知道这些结果返回才执行一些操作。完美地，我更喜欢像这样写：

```
urls.RunAsync(
    url => startDownload(url), task => finishDownload(task.AsyncState.ToString(), task.Re
```

这里使用 startDownload() 方法开始每个 URL 的下载。随着每个下载完成，finsishDownload() 会被执行。一旦所有下载完成， RunAsync() 就完成。 Parallel Task 库完成这个自然做了很多工作，所以我们仔细检查下。最好开始的地方是 RunAsync 方法：

```
public static void RunAsync<T, TResult>( this IEnumerable<T> taskParms, Func<T, Task<TRes
{
    taskParms.Select(parm => taskStarter(parm)).AsParallel().ForAll(t => t.ContinueWith(t
}
```

这个方法为每个输入参数创建一个任务。Select() 方法返回任务序列。下一步，你使用 AsParallel() 并行处理结果。对于每个单一的任务，你会调用后续处理方法。 Task<T> 类表示一个任务，并包含这个任务输入和输出值的属性。 ContinueWith() 是 Task<T> 的其中一个方法。它会在任务完成后调用，允许你对已完成的任务进行处理。在 RunAsync 方法中，它传入 Task 对象参数调用 taskFinisher 。ForAll() 使用 Inverted Enumeration 算法，它会阻塞知道所有任务完成。

我们深入探讨这个模式，理解开始下载方法和汇报下载完成的方法。 finishDownload 方法很简单，我在完成时输出：

```
private static void finishDownload(string url, byte[] bytes)
{
    Console.WriteLine("Read {0} bytes from {1}", bytes.Length, url);
}
```

StartDownload 比 Parallel Task 库的接口复杂一些。具体的类型用来帮助支持 Task 接口。我更想抽象出来，但是针对具体的任务的处理不同类型会有点区别。实际上， Parallel Task 库为 .NET BCL 在此版本之前的很多不同异步模式之上提出了一套通用的接口。

```
private static Task<byte[]> startDownload(string url)
{
    var tcs = new TaskCompletionSource<byte[]>(url);
    var wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) =>
    {
        if (e.UserState == tcs)
        {
            if (e.Cancelled)
                tcs.TrySetCanceled();
            else if (e.Error != null)
                tcs.TrySetException(e.Error);
            else
                tcs.TrySetResult(e.Result);
        }
    };
    wc.DownloadDataAsync(new Uri(url), tcs);
    return tcs.Task;
}
```

这个方法混合了 Task 代码和从 URL 下载的代码，所以我们必须非常认真地梳理一遍。首先，它为这个任务创建一个 TaskCompletionSource 对象。TaskCompeletionSource 对象使得任务的创建和完成分离了。这里是非常重要的，因为你使用 WebClient 类的异步方法创建这个任务。TaskCompletionSource 的参数是这个任务的返回的结果。

WebClient 类使用基于事件的异步模式（EAP）。这意味着你像一个事件注册处理器，当这个异步操作完成时，事件就会被触发。当事件触发 startDownload() 吧任务完成信息存储在 TaskCompletionSource 中。TaskSheduler 选择一个任务并开始下载。这个方法返回的 Task 对象内嵌在 TaskCompletionSource 中，这样当任务完成时事件结果就会被处理。

在这些工作后，web 下载另一个线程异步开始。当下载完成， DownloadDataCompleted 事件就会被触发。事件处理会设置 TaskCompletionSource 的完成状态。在 TaskCompletionSouce 嵌入 Task 对象表示它已经完成。

现在，任务会调用 ContinueWith()，报告下载的结果。花了点功夫解开这些细节，但是在接解开过一次后，这个模式就不会那么难理解。

上面展示的例子就是底层使用的基于事件异步模式的正确表达习惯。 .NET 库的其他领域使用异步编程模型（ APM） 模式。在这个模式，一些操作 Foo 你调用 BeginFoo()，会返回一个 IAsyncResult 对象。一旦操作完成，你可以调用 EndFoo()，传入参数就是这个 IAsyncResult 对象。Parallel Task 库你可以使用 Task<TResult>.Factory.FromAsync() 方法实现这个模式。

底层的原理和我下载 web 数据的版本是类似的。区别在于你提供了不同的委托去匹配使用的异步方法来创建任务。

Parallel Task 库提供一系列方法，使得 I/O 受限制操作和 CPU 受限制一样工作。使用 Task 类，你可以对 I/O 受限制操作或混合了 I/O 和 CPU 受限制操作支持各种异步模式。并行任务还是不那么简单，但是 Parallel Task 库和 PLINQ 比之前的库提供更好的语言层次对异步编程的支持。随着我们程序会更多访问不同机器的数据和更多线程等待远程机器的响应，这会变得更重要。

小结：

一般的多线程都是指的是 CPU 受限制的操作进行并行优化，对于 I/O 也同样存在现在，要等到外部事件响应，PLINQ 和 Parallel Task 库对 I/O 受现在操作也像 CPU 受限制一样提供支持。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2088750

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则37：构造并行算法的异常考量

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

前面两个原则幸福地忽略了任何子线程运行出错的可能性。这显然不是现实世界所进行的。异常会在你的子线程发生，你不得不转向去收拾残局。当然，后台线程的异常在某种程度上增加复杂性。异常不能继续调用线程边界的函数栈。而是，如果在线程启动方法出现异常，这个线程就会终止。没有任何方式调用线程检索错误，或者对异常做任何事。更重要的是，如果出现异常你的并行算法就必须支持回滚，你不得不理解异常出现的副作用并且能从异常恢复过来。每个算法都有不同需求，所以在并行的环境中没有通用的答案处理异常。我只能提供的指导就是：对于特定的应用你可以使用最好的策略。

我们从上个原则的异步下载开始说。最简单策略就是没有副作用，并且从所有的 web 主机上持续下载不用考虑其中一个会失败。并行操作使用新的 AggregateException 类处理并行操作的异常。 AggregateException 的 InnerException 属性包含所有在并行操作可能产生的异常。在这个过程有好几个方式处理这个异常。首先，我会演示一个更普遍的情况，怎么在外部处理子任务产生的错误。

在上一个原则的 RunAsync() 方法在多个并行操作使用。这意味你要捕获 AggregateException 的 InnerException 集合的异常。越多并行操作，嵌套就越深。因为并行操作有不同的构成，你应该防止多次复制在异常集合的元素异常。我修改 RunAsync() 以处理可能的错误：

```
try
{
    urls.RunAsync( url => startDownload(url), task => finishDownload(task.AsyncState.ToSt
}
catch (AggregateException problems)
{
    ReportAggregateError(problems);
}
private static void ReportAggregateError(AggregateException aggregate)
{
    foreach (var exception in aggregate.InnerExceptions)
        if (exception is AggregateException)
            ReportAggregateError( exception as AggregateException);
        else
            Console.WriteLine(exception.Message);
}
```

ReportAggregateError 输出所有不是 AggregateException 自身异常的信息。当然，这会掩盖所有异常，不管你是否有没有预料到。这是相当危险的。相反，你需要处理你可以从中恢复的异常，或者重抛出其他异常。

这里有很多集合递归，所以有一个试用的函数式很有意义的。泛型方法必须知道哪些异常类你要处理，哪些异常你是不期望的并且你要处理那些你想要处理的异常。你需要为这个方法确定要处理的异常类型和处理异常的代码。这是简单的类型和 Action<T> lambda 表达式的字典。并且，如果处理没有处理 InnerException 集合的每个异常，清楚哪些异常出现异常。这说明你要重新抛出原来的异常。下面是新的 Runsync 代码：

```
try
{
    urls.RunAsync(url => startDownload(url), task => finishDownload(task.AsyncState.ToStr
}
catch (AggregateException problems)
{
    var handlers = new Dictionary<Type, Action<Exception>>();
    handlers.Add(typeof(WebException),ex => Console.WriteLine(ex.Message));
    if (!HandleAggregateError(problems, handlers))
        throw;
}
```

HandleAggregateError 方法递归查看每个异常。如果异常是预料的，处理器会被调用。否则， HandleAggregateError 返回 false，说明这类异常不能被正确处理：

```
private static bool HandleAggregateError(AggregateException aggregate, Dictionary<Type, A
{
    foreach (var exception in aggregate.InnerExceptions)
        if (exception is AggregateException)
            return HandleAggregateError( exception as AggregateException, exceptionHandle
        else if (exceptionHandlers.ContainsKey( exception.GetType()))
        {
            exceptionHandlers[exception.GetType()] (exception);
        }
        else
            return false;
    return true;
}
```

这点代码看着有些密集，但是并不难。当它传入一个 AggregateException，它会对子列递归评估。当遍历到任何其他异常，它会查询字典。如果处理器 Action<> 已经被注册，就会调用这个处理器。如果没有，就会理解返回 false，即发现一个不能处理的异常。

你会奇怪为什么当没有注册处理器抛出的是元素的 AggregateException 而不是单一的异常。抛出集合中的单一异常会丢失重要信息。 InnerException 可能包含很多异常。可能会有多个异常是没有预料到的。你必须返回这个集合而避免丢失太多信息。很多情况，AggregateException 的 InnerException 集合只有一个异常。然而，你不能那样写代码因为当你想要额外的信息，它却不在那。

当然，这会感觉有一点丑陋。还有没有更好的防止异常出现使得任务离开运行的后台工作的办法。在几乎所有情况，这是更好的。修改代码使得正在运行的后台任务确保没有异常能停止这个后台任务。当你使用 TaskCompletionSource<> 类，就说明你没有调用 TrySetException()，而是确保每个任何调用 TrySetResult() 表示完成。这就有了下面对 startDownload() 的修改。但是，正如和我前面说的，你不能只是捕获每个异常。你应该只捕

获可以从中恢复的异常。在这个例子中，你可以从 WebException 恢复，这个异常出现因为远程主机不可访问。其他异常类型可能表明更严重的问题。那些会持续产生的异常会终止所有处理。 startDownload 方法有了下面的修改：

```
private static Task<byte[]> startDownload(string url)
{
    var tcs = new TaskCompletionSource<byte[]>(url);
    var wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) =>
    {
        if (e.UserState == tcs)
        {
            if (e.Cancelled)
                tcs.TrySetCanceled();
            else if (e.Error != null)
            {
                if (e.Error is WebException)
                    tcs.TrySetResult(new byte[0]);
                else
                    tcs.TrySetException(e.Error);
            }
            else
                tcs.TrySetResult(e.Result);
        }
    };
    wc.DownloadDataAsync(new Uri(url), tcs);
    return tcs.Task;
}
```

WebException 的返回说明0字节数组读取，而且所有其他异常会通过正常的通道抛出。对的，也就是说当 AggregateException 被抛出仍可以继续对正在发生进行处理。很可能你只是把它们当做致命错误，而你的后台任务可以继续处理其他错误。但是你需要理解所有不同类型的异常。

当然，如果你使用 LINQ 语法，后台任务的错误有引起其他问题。记得原则35我描述了三条和并行算法的区别。在所有情况下，使用 PLINQ 和正常的懒评估会有些变化，而这些变化是你在 PLINQ 算法处理异常时必须考虑的。请记住，通常，一个查询只有在其他代码请求这个查询产生的项时才执行。这当然不是 PLINQ 的工作。后台线程运行产生结果，而且另一个任务组合最后的结果序列。它不能立即评估。查询的结果不是立即产生的。然而，后台线程只要调用运行就会开始产生结果。现在，你意味着必须改变异常处理的代码。在典型的 LINQ 查询，你可以将使用查询结果的代码放在 try/catch 块内。这不需要包裹定义 LINQ 查询表达式的代码：

```
var nums = from n in data
    where n < 150
    select Factorial(n);
try
{
    foreach (var item in nums)
        Console.WriteLine(item);
}
catch (InvalidOperationException inv)
{
// elided
}
```

一旦涉及 PLINQ，你必须在 tyr/cathc 块中封闭查询的定义。而且，当然，记住如果你使用 PLINQ，你必须捕获 AggregateException 而不是无论你原来期望的是什么异常。不管你使用 Pipelining，Stop&Go，或者 Inverted Enumeration 算法都是正确的。

异常在任何算法中都是复杂的。并行任务引起更多并发症。Parallel Task 库使用 AggregateExceptoion 类持有并行算法排除的所有异常。只要有一个后台线程抛出一个异常，其他后端操作都会被停止。你最好的计划是确保在你的并行任务执行代码时没有任何异常抛出。即使这，其他你没有期望的异常也有可能在某些地方抛出。这意味着你必须处理 AggregateException 以控制线程初始化所有后台工作。

小结：

异常，无论什么时候都要考虑进来，好吧，还没有用到 LINQ 和 PLINQ（工作没有这个需求），暂且没有深入的感受。

跑步去，加油，加油！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持"分享"之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2088794

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

第五章 杂项讨论

原则38：理解动态（Dynamic）的利与弊

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

C# 支持的动态类型为提供了到其他地方的桥梁。这不是意味着鼓励你使用动态语言编程，而是提供了 C# 强静态类型到那些动态类型模型的平滑过渡。

然而，这也不会现在你使用动态类型和其他环境交互。C# 类型可以强制转为为动态对象并当做动态对象。和其他事物一样，把 C# 对象当做动态对象具有两面性有好也也有坏。我们通过一个例子看下发生了什么好的和坏的。

C# 泛型的一个局限是为了方法参数不是 System.Object ，你需要制定约束。而且，约束必须是基类，接口，引用类型，值类型，或存在 public 无参构造函数其中的一个。你不能具体到某些已知的方法。当你需要创建依赖像操作符+这样的泛型方法，这就会是限制。动态调用可以修复这个问题。使用的成员在运行时可访问。下面的方法是将两个动态对象相加，只要在运行时操作符+是可访问的：

```
public static dynamic Add(dynamic left, dynamic right)
{
        return left + right;
}
```

这是第一次讨论动态，我们看下这会发生什么。动态可以认为是“ System.Object 的运行时绑定”。在编译时，动态变量只有那些定义在 System.Object 的方法。然而，编译器会增加代码使得每个成员的访问实现为动态寻址调用。在运行时，代码执行检查对象而且决定是否请求的方法是可访问的。（查看原则41实现动态对象。）经常被作为“鸭类型”引用：如果它走起来像鸭子和像鸭子一样说话，它可能就是鸭子。你不需要声明特殊的接口，或者提供任何编译时类型操作。只要成员在运行时可访问，它就会工作。

上面的方法，动态寻址调用会检查两个对象的实际运行时类型是否有操作符+。下面的调用都是正确的：

```
dynamic answer = Add(5, 5);
answer = Add(5.5, 7.3);
answer = Add(5, 12.3);
```

注意 answer 必须声明为动态对象。因为调用是动态的，便器不能知道返回值的类型。它只能在运行时解析。返回值类型要在运行时解析只有声明为动态对象。返回值的静态类型是 dynamic 。它会在运行时解析。

当然，这个动态 Add 方法不光局限于 数字类型。你可以将字符串相加（因为 string 有操作符+的定义）：

```
dynamic label = Add("Here is ", "a label");
```

你还可以将 TimeSpan 和 Date 相加：

```
dynamic tomorrow = Add(DateTime.Now, TimeSpan.FromDays(1));
```

只要操作符+可访问，Add 的动态版本就会工作。

上面开头的解释可能会导致过度使用动态编程。我还只是讨论了动态编程的优点。是时候也该考虑下缺点。你已经抛弃了类型系统的安全，这样，你也就限制了编译器对你的帮助。任何解释类型的错误都只会在运行时才被发现。

任何有一个操作数（包括可能的 this 引用）的操作的结果是动态的即本身是动态的。有时，你会把动态对象转换为你最多使用的静态类型。这就需要强制类型转换或转换操作：

```
answer = Add(5, 12.3);
int value = (int)answer;
string stringLabel = System.Convert.ToString(answer);
```

强制类型转换操作只有当动态对象的实际类型是目标类型或可以转为为目标类型才会工作。你需要知道任何动态操作的正确类型，才能给它强类型。否则，转换就会在运行时失败，并且抛出异常。

当你不知道类型但又不得不在运行时解析方法，动态类型就是正确的工具。当你知道编译时类型，你可以使用 lambda 表达式和函数编程构造你需要的解决方案。你可以使用 lambda 表达式重写 Add 方法：

```
public static TResult Add<T1, T2, TResult>(T1 left, T2 right,Func<T1, T2, TResult> AddMet
{
          return AddMethod(left, right);
}
```

每个调用都需要提供具体的方法，所有前面的例子都可以使用这个策略实现：

```
var lambdaAnswer = Add(5, 5, (a, b) => a + b);
var lambdaAnswer2 = Add(5.5, 7.3, (a, b) => a + b);
var lambdaAnswer3 = Add(5, 12.3, (a, b) => a + b);
var lambdaLabel = Add("Here is ", "a label",(a, b) => a + b);
dynamic tomorrow = Add(DateTime.Now, TimeSpan.FromDays(1));
var finalLabel = Add("something", 3,(a,b) => a + b.ToString());
```

你可以看到最后一个方法需要具体将 int 转换为 string 。它比其他 lambda 方法更不优雅。不幸的是，只有这样方法才能工作。你不得不在应用 lambda 的地方推断出类型。这意味着相当不部分的代码看起来跟手动一样重复因为代码对于编译器是不一样的。当然，定义并实现

Add 方法看起来很傻。实践中，你使用的 lambda 方法不会简单地执行。 .NET 类库使用 Enumerable.Aggregate().Aggregate() 枚举整个队列并计算相加（或者执行其他操作）的结果：

```
var accumulatedTotal = Enumerable.Aggregate(sequence, (a, b) => a + b);
```

这仍看起来你是在重复代码。避免这样重复代码是使用表达式树。另一种运行时编译代码。System.Linq.Expression 类和它的子类提供 API ，你可以构建表达式树。如果你构建表达式树，你会将它转化为 lambda 表达式，然后编译最后的 lambda 表达式为委托。例如，下面这段构建并执行三个类型值的相加：

```
// Naive Implementation. Read on for a better version
public static T AddExpression<T>(T left, T right)
{
    ParameterExpression leftOperand = Expression.Parameter( typeof(T), "left");
    ParameterExpression rightOperand = Expression.Parameter( typeof(T), "right");
    BinaryExpression body = Expression.Add( leftOperand, rightOperand);
    Expression<Func<T, T, T>> adder = Expression.Lambda<Func<T, T, T>>(body, leftOperand,
    Func<T, T, T> theDelegate = adder.Compile();
    return theDelegate(left, right);
}
```

最有趣的的工作涉及类型信息，而不是使用 var ,为了让代码清晰，我特别命名所有的类型。

开始两行为类型 T 的变量“left”，“right”创建了参数表达式。下面两行就是使用这两个参数创建 Add 表达式。 Add 表达式继承自 BinaryExpress 。你也可以创建其他类似的二元操作符。

下面，你需要将带有两个参数的表达式构建为 lambda 表达式。最后你编译并创建 Func<T,T,T> 委托。一旦编译好，你就可以执行它并返回结果。当然，你可以像其他泛型方法一样调用这个方法：

```
int sum = AddExpression(5, 7);
```

添加在上面例子的注释说明这是朴素的实现。不要复制这段代码到你应用中。这个版本有两个问题。第一，有很多情况 Add() 可以工作但这个却不可以。有很多例子的 Add() 方法的参数是不同的： int 和 double ， DateTime 和 TimeSpan 等等。这些情况这个方法都不能工作。我们对此进行修复。你必须再增加两个泛型参数。然后，你可以指定左右两个参数为不同类型。同时，我用 var 声明一些局部变量。这掩盖了类型信息，但它确实帮助使逻辑的方法更清晰。

```
// A little better.
public static TResult AddExpression<T1, T2, TResult> (T1 left, T2 right)
{
    var leftOperand = Expression.Parameter(typeof(T1), "left");
    var rightOperand = Expression.Parameter(typeof(T2), "right");
    var body = Expression.Add(leftOperand, rightOperand);
    var adder = Expression.Lambda<Func<T1, T2, TResult>>( body, leftOperand, rightOperand
    return adder.Compile()(left, right);
}
```

这个版本跟前面的很类似；它只是可以让你调用左右参数不同的类型。唯一的缺点是你需要指定三个参数的类型，无论你怎么调用：

```
int sum2 = AddExpression<int, int, int>(5, 7);
```

因为你指定三个不同参数的类型，不同类型表达式就可以工作：

```
DateTime nextWeek= AddExpression<DateTime, TimeSpan,DateTime>(DateTime.Now, TimeSpan.From
```

是时候该解决让人烦恼的问题了。我前面展示的代码，每次你调用 AddExpression() 方法就会编译表达式为委托。这是相当低效的，特别是你重复执行相同的表达式。编译表达式是非常耗性能的，所以你应该为你后面的调用缓存编译好的委托。下面是这个类的第一个初稿：

```
// dangerous but working version
public static class BinaryOperator<T1, T2, TResult>
{
    static Func<T1, T2, TResult> compiledExpression;
    public static TResult Add(T1 left, T2 right)
    {
        if (compiledExpression == null)
            createFunc();
        return compiledExpression(left, right);
    }
    private static void createFunc()
    {
        var leftOperand = Expression.Parameter(typeof(T1), "left");
        var rightOperand = Expression.Parameter(typeof(T2), "right");
        var body = Expression.Add(leftOperand, rightOperand);
        var adder = Expression.Lambda<Func<T1, T2, TResult>>(body, leftOperand, rightOper
        compiledExpression = adder.Compile();
    }
}
```

在这点上，你可能想知道应该使用哪种技术：动态或表达式。这个决定因情况而定。表达式版本更适合简单的计算。很多情况会更快些。而且，表达式比动态调用会少些动态。记得使用动态调用，你可以添加更多不同的类型：int 和 double ， short 和 float 。只要它是合法的 C# 代码，它就是合法的编译的版本。你甚至可以将字符串和数字相加。如果这些情况使用表

达式版本，就会抛出 InvalidOprationException 异常。即使有类型转换，你构建的表达式不会编译为类型转换的 lambda 表达式。动态调用做了更多工作，因此支持更多类型的操作。例如，假设你想更新 AddExpression 添加不同类型和进行适当的转换。好的，你只需要

更新参数和结果类型转换的代码。就是下面这样的：

```
// A fix for one problem causes another
public static TResult AddExpressionWithConversion
<T1, T2, TResult>(T1 left, T2 right)
{
    var leftOperand = Expression.Parameter(typeof(T1),
    "left");
    Expression convertedLeft = leftOperand;
    if (typeof(T1) != typeof(TResult))
    {
        convertedLeft = Expression.Convert(leftOperand, typeof(TResult));
    }
    var rightOperand = Expression.Parameter(typeof(T2),"right");
    Expression convertedRight = rightOperand;
    if (typeof(T2) != typeof(TResult))
    {
        convertedRight = Expression.Convert(rightOperand, typeof(TResult));
    }
    var body = Expression.Add(convertedLeft, convertedRight);
    var adder = Expression.Lambda<Func<T1, T2, TResult>>(body, leftOperand, rightOperand)
    return adder.Compile()(left, right);
}
```

这就修复了像 double 和 int 相加，或 string 加上 double 返回 string 的需要转换的问题。然而，当参数和结果不相同的使用就变得无效了。特别地，这个版本针对上面 TimeSpane 和 DataTiem 的例子就不能工作。添加更多的代码，你就可以修复这个问题。然而，在这点上，你已经很漂亮实现 C# 动态调度的代码（查看原则41）。使用动态，而不是做所有更多工作。

当操作数和结果的类型是一样的时候，你应该使用表达式版本。你使用泛型参数接口，并且更少的情况会在运行时失败。下面的版本是我推荐在运行时调度使用的表达式版本的实现：

```
public static class BinaryOperators<T>
{
    static Func<T, T, T> compiledExpression;
    public static T Add(T left, T right)
    {
        if (compiledExpression == null)
            createFunc();
        return compiledExpression(left, right);
    }
    private static void createFunc()
    {
        var leftOperand = Expression.Parameter(typeof(T), "left");
        var rightOperand = Expression.Parameter(typeof(T), "right");
        var body = Expression.Add(leftOperand, rightOperand);
        var adder = Expression.Lambda<Func<T, T, T>>(body, leftOperand, rightOperand);
        compiledExpression = adder.Compile();
    }
}
```

当你调用 Add 时，你仍需要指定一个参数的类型。这么做的优势是编译器可以在调用时进行类型转换。编译器会将 int 提升为 double ，等。

使用动态和运行时构建表达式都会耗性能。和任何动态类型系统一样，你的程序在运行时需要做更多工作，因为编译器没有执行检查使用的类型。编译器必须在产生在运行时检查的指令。我并不打算夸大，因为 C# 编译器产生高效的运行时检查类型的代码。很多情况下，使用动态会比你自己写的反射来产生晚绑定更快。然而，运行时的工作量是不可忽略的，它花费的时间也是不能忽略的。如果你可以使用静态类型解决这个问题，那毫无疑问比使用动态类型更高效。

当你掌握所有涉及的类型，你可以创建接口而不是使用动态编程，那是更好的解决方案。你可以定义接口，面向接口编程，并且让所有类实现这个接口就可以有接口定义的行为。 C# 类型系统会严格检查引入的类型错误，而且编译器会产生更高效的代码，因为它可以假定某些类型的错误是不可能出现的。

很多情况，你可以使用泛型 lambda 创建泛型 API 并且强制调用者定义在动态算法执行的代码。

第二个选择是使用表达式。如果你类型的组合情况相对较少和很少的类型转换，这是合适的选择。你可以控制表达式的创建，因此控制运行时的开销。

当你使用动态，底层的动态实现会尽可能使构造工作合法，无论运行花费多大的消耗。

然而，我在开头演示的 Add() 方法，是不完善的。 Add() 应该能对 .NET 类框架定义的很多类型工作。你不能往回并添加 IAdd 接口到这些类型。你也不能保证你所有使用的第三方类库符合这个新接口的功能。构建基于已存在类型的特定成员的方法的最好的方式是编写动态方法并且在运行时推断具体的选择。动态实现要找到一个恰当的实现，使用并缓存有助于更好的性能。它比单纯的静态类型解决方法更耗时，却比表达式树的解析更简单。

小结：

这个原则是第五章的开头，介绍了动态的利与弊，比较泛，没有提多精华，还是得看后面的几篇原则。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2090155

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则39：使用动态对泛型类型参数的运行时类型的利用

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

System.Linq.Enumerable.Cast<T> 强制序列的每个对象转化为目标类型 T 。这是框架的一部分，所以 LINQ 查询中的 IEnumerable （而不是 IEnumerable<T> ）才能使用。 Cast<T> 是一个没有约束的泛型方法。这就是限制类型转换使用它。如果你理解 Cast<T> 的这个限制，你会发现你自己想的却不能工作。现实中，它就是该本来那样工作，而不是你期望的那样。我们检查他们的内部工作和限制。然后，你就可以很容易创建不同的版本并像你期望一样。

问题的根源在于，事实上 Cast<T> 在没有任何 T 的知识的清下编译成 MISL，所以 T 必须是继承于 System.Object 。因此，它只能使用定义在 System.Object 的函数。查看下面的这个类：

```
public class MyType
{
    public String StringMember { get; set; }
    public static implicit operator String(MyType aString)
    {
        return aString.StringMember;
    }
    public static implicit operator MyType(String aString)
    {
        return new MyType { StringMember = aString };
    }
}
```

查看原则28就知道为什么转换操作符那么不好；然而，用户定义的转换操作符就是这个问题的关键。考虑这段代码（假设 GetSomeStrings() 返回字符串序列）：

```
var answer1 = GetSomeStrings().Cast<MyType>();
try
{
    foreach (var v in answer1)
        Console.WriteLine(v);
}
catch (InvalidCastException)
{
    Console.WriteLine("Cast Failed!");
}
```

在开始遍历元素之前，你可能希望 GetSomeStrings().Cast<MyType>() 已经使用定义在 MyType 的隐式转换操作符对每个 string 正确转换为 MyType 。现在你知道它并没有，它会抛出 InvalidCastException 。

上面的代码和下面使用查询表达式是一样的构造：

```
var answer2 = from MyType v in GetSomeStrings()
    select v;
try
{
    foreach (var v in answer2)
        Console.WriteLine(v);
}
catch (InvalidCastException)
{
    Console.WriteLine("Cast failed again");
}
```

在循环体声明的变量被编译器调用 Cast<MyType> 。同样，它会抛出 InvalidCastException 。以下面的方式进行重构，就能工作：

```
var answer3 = from v in GetSomeStrings()
    select (MyType)v;
foreach (var v in answer3)
    Console.WriteLine(v);
```

有什么不同？前面的两个版本使用 Cast<T>() 不能工作，这个版本能工作是因为 lambda 的 Select() 的参数使用了强制类型转换。 Cast<T> 不可能在运行时访问参数类型任何自定义的转换操作符。只有引用转换或封箱转换才能使用转换操作符。只有当 is 操作符成功时，引用转换才会成功（查看原则3）。封箱转换是将值类型转换引用类型反之亦然（查看原则45）。Cast<T> 不能访问用户定义的转换操作符因为它只能假设 T 包含定义在 System.Object 的成员。System.Object 没有包含任何自定义的转换操作符，所以这些都是不合格的。使用 Select<T> 成功时因为 lambda 使用 Select() 的输入参数类型是 string 。这就能使用定义在 MyType 的转换操作符。

我已经在前面指出过，我常把转换操作符看着是代码的小菜。偶尔，它们是有用的，但是静态是引起比它们的价值更多的问题。如果没有转换操作符，这里就不会有开发者写些不能工作的示例代码。

当然，如果我建议不要使用转换操作符，我应该会给一个替代的解决方案。 MyType 已经包含可读/写的存储 string 的属性，所以我只是把转换操作符去掉并使用构造函数：

```
var answer4 = GetSomeStrings().
Select(n => new MyType { StringMember = n });
var answer5 = from v in GetSomeStrings()
select new MyType { StringMember = v };
```

这样，我只需要为 MyType 创建不同的构造函数。当然，这只是针对 Cast<T> 的限制。既然你已经明白为什么这些限制会存在，现在就该写一个不同方法绕过这个限制。诀窍就是编写一个泛型方法并利用运行时形象进行类型转换。

你可能写了一页又一页基于反射的代码以检查哪些转换是可用的，并执行那些转换返回恰当的类型。你可以那样做，但是这是浪费。相反， C#4.0 ，动态做了所有繁重的活。你只需要简单 Convert<T> 就可以得到你期望的：

```
public static IEnumerable<TResult> Convert<TResult>( this System.Collections.IEnumerable
{
    foreach (object item in sequence)
    {
        dynamic coercion = (dynamic)item;
        yield return (TResult)coercion;
    }
}
```

现在，只要从源类型到目标类型的转换（隐式或显示），转换就会工作。仍然有强制类型转换会涉及到，所有运行时失败还是有可能发生。Convert<T> 与 Cast<T> 的比较中，ConverT<T> 适用于更多的类型转换的情况，但是它需要做更多工作。作为一个开发者，你应该更多关心用户需要我们创建什么代码而不只是我们自己的代码。 Convert<T> 可以通过整个测试：

```
var convertedSequence = GetSomeStrings().Convert<MyType>();
```

Cast<T> 像其他泛型方法一样，编译时对参数类型只有有限掌握。这会导致泛型方法不能像你期望一样工作。根本原因就是泛型方法不知道类型参数代表的类型的特定的功能。当这种情况发生时，一个小的应用的动态可使运行时反射使问题得到解决。

小结：

这个原则很简单却很使用（trick），作者给的建议是：使用泛型参数类型对动态对象进行强制类型转换。

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com）交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2090174

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）

原则40：使用动态接收匿名类型参数

By D.S.Qiu

尊重他人的劳动，支持原创，转载请注明出处：**http://dsqiu.iteye.com**

匿名参数的一个缺陷是你不能轻易让一个方法使用其作为参数或返回值。因为编译器产生的匿名类型，你不能用来作为方法的参数或返回值。这个问题的任何解决方案都是与局限的。你可以使用匿名类型作为泛型类型参数，或者传给参数为 System.Object 的方法。这些都不会觉得特别满意。泛型方法只能假设类型定义在 System.Object 的功能。 System.Object 同样也有局限。当然，有些时候，你会发现确实需要对一个实际的行为的类命名。本节主要讨论当你需要使用不同的匿名类型，即他们有相同命名的属性但不是你应用的核心部分，而且你不想创建新的命名类型时，你需要做什么。

动态类型可以让你克服这个限制。动态是在运行时绑定并且使编译器为可能的运行时类型生成必需的代码。

假设你需要打印一个价格表的信息。进一步假设你的价格表，可以从多个数据源的产生。你可能有一个存储仓库的数据库，另一个存储特别订单，并且还有一个是存储通过第三方供应商出售的价格。因为它们都是不同的系统，它们可能是不同商品的抽象。这些不同的抽象可能没有相同名字的属性，而且它们没有共同的基类或实现同一个接口。经典的解放方案是实现一个适配器模式（查看 Design Patterns, Gamma, Helm, Johnson, &Vlisside, pp.139-142）为每个产品进行抽象，并转换每个对象为一致的类型。这有相当多的工作，你要为每个新产品的抽象添加适配器。而且，适配器模式在静态类型系统会有更好的性能。

另一个，轻量的解决方案是是用动态创建方法，传入有任何价格信息的类型：

```
public static void WritePricingInformation(dynamic product)
{
    Console.WriteLine("The price of one {0} is {1}", product.Name, product.Price);
}
```

你可以创建匿名类型，通过在你的价格方法中匹配属性，就可以从数据源得到价格信息：

```
var price = from n in Inventory
where n.Cost > 20
select new { n.Name, Price = n.Cost * 1.15M };
```

你任何项目的动态方法创建匿名类型都要包含必需的属性。只要有命名为“ Price ”和“ Name ”的属性，WritePricingInformation 方法就可以完成这个工作。

当然，你还可以使用匿名类的其他属性。只要属性包含在价格信息中，你就是对的：

```
var orderInfo = from n in Ordered
    select new {
        n.Name,
        Price = n.Cost * 1.15M,
        ShippingCost = n.Cost / 10M
    };
```

普通的命名的 C# 对象可以在动态中使用。这说明你的价格信息方法可以使用含有正确命名的属性的具体的类：

```
public class DiscountProduct
{
    public static int NumberInInventory { get; set; }
    public double Price { get; set; }
    public string Name { get; set; }
    public string ReasonForDiscount { get; set; }
    // other methods elided
}
```

你可能主要到 DiscountProduct 的 Price 属性的类型是 double 而之前的匿名类型的 Price 的类型是 decimal 。这样也是可以的。 WritePricingInformation 使用动态静态类型，所以它会在运行时确定是否正确。当然，如果 DiscountProduct 继承自 Product 类，并且 Product 类包含 Name 和 Price 属性，也是可以工作的。

上面的代码可能会让你很容易相信我比我实际中更主张使用动态。动态调用的确是解决这个问题的好方法，但不要过度使用。动态调用意味着你需要支付额外的开销。当你需要时，这个开销是值得的，但是当你可以避免它，你就应该避免。

你不得不在静态和动态调用之间做出选择。你可以创建 WritePricingInformation() 方法的重写，就可以具体到你对象模型的每个产品类：

```
public class Product
{
    public decimal Cost { get; set; }
    public string Name { get; set; }
    public decimal Price
    {
        get { return Cost * 1.15M; }
    }
}
// Derived Product class:
public class SpecialProduct : Product
{
    public string ReasonOnSpecial { get; set; }
    // other methods elided
}
// elsewhere
public static void WritePricingInformation(dynamic product)
{
    Console.WriteLine("The price of one {0} is {1}", product.Name, product.Price);
}
public static void WritePricingInformation(Product product)
{
    Console.WriteLine("In type safe version");
    Console.WriteLine("The price of one {0} is {1}",product.Name, product.Price);
}
```

编译器会针对 Product 或 SpecialProduct 的对象（或者对象模型中任何其他子类的对象）使用具体的版本。对于其他类型，编译器静态类型版本当做动态使用。这包括匿名类型。动态绑定器在内部会将每个使用的方法缓存起来。类似调用 WritePricingInformation() 一样反复调用同一匿名类型的情况，可以减少开销。一旦方法在第一次调用绑定，就会可以在后续的调用重用。这是零开销，但动态的实现应该尽可能最小化使用动态的开销。

你可能怀疑为什么这些方法不能是扩展方法，这样就可以看起来是匿名类型的成员。的确，那是很不错的想法，但这在 C# 是不合法的。你不允许创建多态对象的扩展方法。

你可以利用动态创建使用匿名类型的方法。要像刺激的调味品一样少用这项技术。如果你发现你为了使用匿名类型使用动态调用创建很多方法，这很明显的迹象，你需要创建具体的类型来表示这些概念。这更容易日后维护，你也会得到类型系统和编译器的更多支持。然而，当你需要一到两个使用匿名类型的实用方法，动态调用是创建这个行为的简单的方法。

小结：

哎，作者一再强调动态耗时，但是一直没有具体指出哪些过程为什么耗时！

欢迎各种不爽，各种喷，写这个纯属个人爱好，秉持”分享“之德！

有关本书的其他章节翻译请点击查看，转载请注明出处，尊重原创！

如果您对D.S.Qiu有任何建议或意见可以在文章后面评论，或者发邮件（gd.s.qiu@gmail.com)交流，您的鼓励和支持是我前进的动力，希望能有更多更好的分享。

转载请在文首注明出处：http://dsqiu.iteye.com/blog/2090600

更多精彩请关注D.S.Qiu的博客和微博（ID：静水逐风）